# TP NoSQL
# Redis

Nom : BOUJMIL
Prénom : Zeineb

Redis est une base de données clé/valeur en mémoire : chaque clé identifie une valeur qui peut être une chaîne, une liste, un set, un hash, etc.
On peut :
**Créer (C)** une donnée : SET
**Lire (R)** : GET
**Mettre à jour (U)** : refaire SET sur la même clé
**Supprimer (D)** : DEL
C'est l'implémentation la plus simple du **CRUD** dans Redis .

### 1. Lancement de Redis et connexion

```
PS C:\WINDOWS\system32> docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
              NAMES
135e858406be   redis:latest   "docker-entrypoint.s…"   10 hours ago   Up 10 hours   0.0.0.0:6379->6379/tcp, [::]:6379->6
379/tcp       redis
a29682e01cc9   mongo:4.2.24   "docker-entrypoint.s…"   10 hours ago   Up 10 hours   27017/tcp
              adoring_pike
8402ab127373   mongo:4.2.24   "docker-entrypoint.s…"   10 hours ago   Up 10 hours   0.0.0.0:27017->27017/tcp, [::]:27017
->27017/tcp   mongodb
PS C:\WINDOWS\system32> docker exec -it 135e858406be redis-cli
```

Cela lance le **client Redis** à l'intérieur du conteneur.
On peut le faire par name aussi et pas par containerID .

```
127.0.0.1:6379>
127.0.0.1:6379> set demo "Bonjour"
OK
127.0.0.1:6379> set user:1234 "Zeineb"
OK
127.0.0.1:6379> get user:1234
"Zeineb"
127.0.0.1:6379> del user:1234
(integer) 1
127.0.0.1:6379> del user:1234
(integer) 0
127.0.0.1:6379> set 27nov 0
OK
127.0.0.1:6379> incr 27nov
(integer) 1
127.0.0.1:6379> incr 27nov
(integer) 2
127.0.0.1:6379> incr 27nov
(integer) 3
127.0.0.1:6379> incr 27nov
(integer) 4
127.0.0.1:6379> incr 27nov
(integer) 5
127.0.0.1:6379> decr 27nov
(integer) 4
127.0.0.1:6379> decr 27nov
(integer) 3
```

La clé **27nov** est utilisée comme compteur.
**INCR** augmente la valeur entière de 1, **DECR** la diminue de 1.
Redis convertit automatiquement la valeur de 27nov en entier.

Comme les données sont en RAM, on ne peut pas tout garder **indéfiniment** car la RAM est limitée et certaines informations n'ont du sens que pendant un certain temps (sessions, codes, cache).
Donc Redis permet de fixer une **durée de vie (TTL)** sur une clé :

```
127.0.0.1:6379> set macle maval
OK
127.0.0.1:6379> ttl macle
(integer) -1
127.0.0.1:6379> expire macle 120
(integer) 1
127.0.0.1:6379> ttl macle
(integer) 118
127.0.0.1:6379> del macle
(integer) 1
```

**EXPIRE key seconds :** associe un timeout à une clé. Après ce délai, Redis supprime la clé automatiquement.
**TTL key :** renvoie le temps restant en secondes ; -1 si la clé n'a pas d'expiration, -2 si la clé n'existe pas.

Au début TTL retourne -1 : la clé existe mais n'a pas de durée de vie.
Après EXPIRE macle 120, TTL renvoie un nombre positif (118 car le temps a déjà passé ) : la clé est volatile, elle sera effacée automatiquement après 120 secondes.

# Les listes

```
127.0.0.1:6379> rpush mesCours "BDA"
(integer) 1
127.0.0.1:6379> rpush mesCours "noSql"
(integer) 2
127.0.0.1:6379> lrange mesCours 0 -1
1) "BDA"
2) "noSql"
127.0.0.1:6379> lrange mesCours 0 0
1) "BDA"
127.0.0.1:6379> lrange mesCours 1 1
1) "noSql"
127.0.0.1:6379> lpop mesCours
"BDA"
127.0.0.1:6379> rpop mesCours
"noSql"
127.0.0.1:6379> lrange mesCours 0 -1
(empty array)
```

**RPUSH , LPUSH** ajoutent des éléments respectivement en fin et debut de liste.
**LRANGE** 0 -1 récupère toute la liste . les deux indices sont l'indice de la 1ere et la derniere valeur retournée
**LPOP** et **RPOP** enlèvent et renvoient respectivement le premier et le dernier élément

Une liste est une séquence ordonnée d'éléments. Elles acceptent les valeurs redondantes.
C'est utile pour représenter une file d'attente, un historique où le même élément peut revenir plusieurs fois, etc.

```
127.0.0.1:6379> rpush mesCours "BDA"
(integer) 1
127.0.0.1:6379> rpush mesCours "BDA"
(integer) 2
127.0.0.1:6379> rpush mesCours "BDA"
(integer) 3
```

# Les sets

Par contre , un set est un ensemble non ordonné d'éléments uniques.
on ne peut pas le supprimer selon son indice , mais avec sa valeur .

```
127.0.0.1:6379> sadd utilisateurs "zeineb"
(integer) 1
127.0.0.1:6379> sadd utilisateurs "marc"
(integer) 1
127.0.0.1:6379> sadd utilisateurs "julien"
(integer) 1
127.0.0.1:6379> sadd utilisateurs "samir"
(integer) 1
127.0.0.1:6379> sadd utilisateurs "samir"
(integer) 0
127.0.0.1:6379> sadd utilisateurs "samir"
(integer) 0
127.0.0.1:6379> sadd utilisateurs "samir"
(integer) 0
```

La deuxième insertion de "samir" est ignorée :
un set ne peut pas contenir de doublons.

C’est très pratique quand on veut savoir *qui* appartient à un groupe sans risque de compter deux fois la même personne

Redis ne garde pas “premier”, “deuxième”, etc.
Il stocke juste un groupe d’éléments uniques.
La seule façon de supprimer est par la valeur, avec SREM

```
127.0.0.1:6379> smembers utilisateurs
1) "zeineb"
2) "marc"
3) "julien"
4) "samir"
127.0.0.1:6379> srem utilisateurs "marc"
(integer) 1
127.0.0.1:6379> smembers utilisateurs
1) "zeineb"
2) "julien"
3) "samir"
```

On crée un autre set :

```
127.0.0.1:6379> sadd autresUtilisateurs "antoine"
(integer) 1
127.0.0.1:6379> sadd autresUtilisateurs "philipe"
(integer) 1
```

On fait son union avec le précédent .

```
127.0.0.1:6379> sunion utilisateurs autresUtilisateurs
1) "antoine"
2) "samir"
3) "zeineb"
4) "philipe"
5) "julien"
```

## Les sets ordonnés

un Set ordonné associe à chaque élément un score numérique et le trie automatiquement selon ce score.

```
127.0.0.1:6379> zadd score4 19 "augustin"
(integer) 1
(1.58s)
127.0.0.1:6379> zadd score4 18 "ines"
(integer) 1
127.0.0.1:6379> zadd score4 10 "samir"
(integer) 1
127.0.0.1:6379> zadd score4 8 "philipe"
(integer) 1
```

score4 est le nom du sorted set.

Chaque élément est stocké sous la forme :
membre + score numérique
augustin → 19
ines → 18
samir → 10
philipe → 8

```
127.0.0.1:6379> zrange score4 0 -1
1) "philipe"
2) "samir"
3) "ines"
4) "augustin"
127.0.0.1:6379> zrevrange score4 0 -1
1) "augustin"
2) "ines"
3) "samir"
4) "philipe"
```

**ZRANGE** renvoie tous les éléments (0 premier, -1 dernier), triés par score croissant (du plus petit au plus grand) : 8 < 10 < 18 < 19 → philipe, samir, ines, augustin

**ZREVRANGE** renvoie les membres en ordre décroissant :
19, 18, 10, 8 → augustin, ines, samir, philipe
C'est exactement ce qu'on utilise pour un classement :
le meilleur score en premier.

```
127.0.0.1:6379> zrank score4 "augustin"
(integer) 3
```

**ZRANK** donne le rang du membre, en comptant à partir de 0
(0 = premier, 1 = deuxième, etc.).
Ici, en ordre croissant, l'ordre est :
philipe (0), samir (1), ines (2), augustin (3)
Donc Redis renvoie 3 → augustin est 4e si on regarde du plus petit score au plus grand.

En pratique, pour un classement du plus grand au plus petit, on utiliserait plutôt ZREVRANK (rang dans l'ordre décroissant).

# Les types Hash

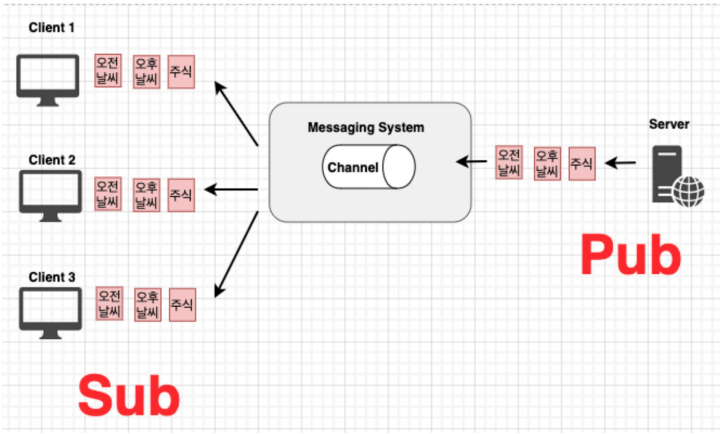
Le type Hash permet de stocker, sous une seule clé, une structure qui ressemble à un objet ou un document (par exemple un profil utilisateur).

```
127.0.0.1:6379> hset user:11 age 24
(integer) 1
127.0.0.1:6379> hset user:11 email zeinebboujmil0@gmail.com
(integer) 1
127.0.0.1:6379> hset user:11 username zeineb
(integer) 1
127.0.0.1:6379> hgetall user:11
1) "age"
2) "24"
3) "email"
4) "zeinebboujmil0@gmail.com"
5) "username"
6) "zeineb"
```

user:11 est la clé principale du hash (l'objet utilisateur).
age, email, username sont des champs (fields).
24, zeinebbojmil10@gmail.com, zeineb sont les valeurs associées à chaque champ.
**HMSET** crée tous les champs d'un coup
**HINCRBY** incrémente le champ age
**HVALS** pour récupérer uniquement les valeurs.

```
127.0.0.1:6379> hmset user:4 username "zeineb" age 24 email zeinebboujmil0gmail.com
OK
127.0.0.1:6379> hgetall user:4
1) "username"
2) "zeineb"
3) "age"
4) "24"
5) "email"
6) "zeinebboujmil0gmail.com"
127.0.0.1:6379> hincrby user:4 age 4
(integer) 28
127.0.0.1:6379> hget user:4 age
"28"
127.0.0.1:6379> hvals user:4
1) "zeineb"
2) "28"
3) "zeinebboujmil0gmail.com"
127.0.0.1:6379>
```

# PUB/SUB




Pub/Sub = Publish / Subscribe = système de messagerie .
Un ou plusieurs clients s'abonnent à un canal .
d'autres clients publient des messages sur ce canal .
Application :
Ouverture d'un deuxième client Redis qui se connecte au même serveur, sur l'adresse 127.0.0.1 et le port 6379 :

```
PS C:\Users\BAZ INFO> docker exec -it 135e858406be redis-cli

127.0.0.1:6379>
```

Maintenant j'ai 2 clients .
**client 1 :**

```
127.0.0.1:6379> subscribe mesCours user:1
1) "subscribe"
2) "mesCours"
3) (integer) 1
1) "subscribe"
2) "user:1"
3) (integer) 2
Reading messages... (press Ctrl-C to quit or any key to type command)
```

le client vient de s'abonner au canal mesCours et user:1. Il joue le rôle de receiver

**client 2 :**

```
127.0.0.1:6379> publish mesCours "un nouveau cours sur no SQL "
(integer) 1
```

le message "un nouveau cours sur noSQL" a bien été publié sur le canal mesCours . Il joue le rôle de publisher

**le client 1 abonné à ce canal a reçu ce message :**

```
3) "un nouveau cours sur no SQL "
```

Quand on exécute PUBLISH mesCours "un nouveau cours sur NoSQL", tous les clients abonnés au canal mesCours reçoivent ce message. La valeur renvoyée par Redis , indique combien de clients abonnés ont reçu le message. Si trois clients sont abonnés à mesCours, Redis renverra (integer) 3 ici , le user ne s'abonne pas à un canal précis, mais à tous les canaux dont le nom commence par **mes** :

```
127.0.0.1:6379> psubscribe mes*
1) "psubscribe"
2) "mes*"
3) (integer) 1
```

publication :

```
127.0.0.1:6379> publish mesnotes "une nouvelle note arrive "
(integer) 1
```

message reçu par le subscriber :

```
1) "pmessage"
2) "mes*"
3) "mesnotes"
4) "une nouvelle note arrive "
```

## Redis : caractéristiques et intérêts

### 1. Stockage en mémoire et rôle de cache

Redis est une base de données en mémoire : les données sont stockées dans la RAM, ce qui rend l'accès beaucoup plus rapide qu'avec une base classique stockée uniquement sur disque. En pratique, Redis est souvent utilisé en complément d'une base de données persistante (MySQL, PostgreSQL, MongoDB, etc.) qui reste la « source de vérité ».

**Fonctionnement typique d'un cache avec Redis :**

- Lorsqu'une ressource est demandée, l'application interroge d'abord Redis.
- Si la clé existe, la réponse est immédiate car elle est lue en RAM.
- Si la clé n'existe pas, l'application va chercher la donnée dans la base « lente » sur disque, puis met le résultat en cache dans Redis pour les prochaines requêtes.

Cette architecture permet de limiter les accès disque (qui travaillent par blocs et restent bien plus lents que la RAM) et d'accélérer fortement les lectures fréquentes.

### 2. Persistance configurables (RDB et AOF)

Même si Redis est avant tout une base en mémoire, il peut rendre ses données persistantes grâce à des mécanismes de sauvegarde configurés dans le fichier *redis.conf*.

**Les deux principaux mécanismes de persistance sont :**

- RDB (snapshots) : Redis effectue régulièrement des sauvegardes complètes de l'état en mémoire dans un fichier binaire, typiquement nommé *dump.rdb*.
- AOF (Append Only File) : chaque commande de modification est ajoutée à un journal, par défaut *appendonly.aof*, ce qui permet de reconstruire l'état de la base au redémarrage.

Grâce à ces options, Redis reste un « in-memory data store » très rapide, tout en étant capable de rejouer ou de restaurer ses données depuis le disque, sans forcément dépendre d'une autre base pour la persistance.

## 3. Modèle NoSQL flexible (sans schéma ni contraintes)

Redis est une base NoSQL : il n'y a pas de schéma fixe comme dans les bases relationnelles (pas de CREATE TABLE avec des colonnes imposées).

### Conséquences :

- Deux utilisateurs peuvent avoir des champs complètement différents dans leurs hashes (par exemple user:1 avec seulement un nom, user:2 avec nom, âge, email, etc.).
- Il n'y a pas de contraintes relationnelles (clé étrangère, NOT NULL, etc.) : la logique métier est gérée au niveau de l'application.

Cette liberté, combinée au fait que Redis est en mémoire et peut être réparti sur plusieurs nœuds (cluster), facilite le passage à l'échelle : on peut ajouter de nouveaux types de données ou de nouveaux nœuds de calcul sans migrations de schéma complexes.