Zunächst erstellen wir uns wieder eine Redis-Instanz über Docker

```
docker network create redis-demo
docker run --name redis-demo --network redis-demo -d redis
```

Und starten einen 2. Container mit dem wir uns zum Redis-Server verbinden.

```
docker run -it --network redis-demo --rm redis redis-cli -h redis-demo
```

REDIS

Redis ist key-value basiert. Das heißt wir speichern uns einen Wert unter einem Key ab.

Set key-value

Um ein Key-Value Paar hinzuzufügen nutzen wir `SET`

```
 SET server:name "fido"
```

Get key-value

Um unseren Key jetzt zu bekommen nutzen wir `GET <key>`

```
GET server:name "fido"
```

Check if exists

Um zu überprüfen ob ein key existiert nutzen wir `EXISTS <key>`
Eine `1` bedeutet der Key existiert. Eine `0` bedeutet er existiert nicht.

```
EXISTS server:name
```

Delete

Mit `DEL <key>` können wir den Key wieder löschen

```
DEL server:name
```

Increment / Decrement

Bei Operationen wir Increment und Decrement kann es zu Problemen kommen, wenn 2 Services gleichzeitig auf die Daten zugreifen wollen. Deshalb bietet Redis die `INCR` und `DECR` Funktionen.

Mit `INCR <key>` oder `DECR <key>` wird der Wert in dem Key um 1 erhöht oder verringert.
Mit `INCRBY <key> <amount>` oder `DECRBY <key> <amount>` um `amount` erhöht oder verringert.
Ein gleichzeitiger Zugriff auf die Funktion ist kein Problem, Redis sorgt dafür, dass alle Increments und Decrements ausgeführt werden.

```
SET connections 10
INCR server:name
```

```
INCRBY connections 100
```

```
SET connections 10
DECR server:name
DECRBY connections 100
```

Vorteil: Ist atomic, keine Probleme mit gleichzeitigem Zugriff auf Daten

Expire / TTL

Wir können einen key-value Paar eine TTL (Time to life) geben. Nach dem ablauf dieser Zeit wird das Paar gelöscht (Nicht mehr über GET erreichbar, bzw. returned nil). Das Kommando lautet `EXPIRE <key> <TTL>`. Solange die TTL noch nicht überschritten ist, können wir mit einem erneuten Aufruf die Zeit erneut setzen. Dies überschreibt dann die derzeit verbleibende Zeit. Die Zeit ist immer in Sekunden.

```
SET resource:lock "Redis Demo"
EXPIRE resource:lock 120
```

Über `TTL <key>` bekommen wir die verbleibende TTL des `key`s

```
TTL resource:lock
```

Dabei heißt: `-1`: Will never expire `-2`: Does not exists anymore `40`: 40 Sekunden TTL verbleibend.

Wir können auch beim setzen des Paares die TTL angeben: `SET <key> <value> EX <TTL>`

```
```

```
SET resource:lock "Redis Demo" EX 120
TTL resource:lock
```

Um ein `EXPIRE` abzubrechen, können wir `PERSIST <key>` nutzen. Das setzt die TTL auf `-1` (will never expire). Expire abbrechen:

```
PERSIST resource:lock
```

Listen

Listen sind wie Arrays. `LLEN`, `LPOP`, `RPOP` Alles was die Liste füllt, erstellt die Liste. Man muss sie nicht erst erstellen, dann füllen. Mit `RPUSH`, `LPUSH`, können wir Elemente einer Liste hinzufügen (Linksseitig, Rechtsseitig).

```
RPUSH friends "Alice"
RPUSH friends "Bob"
RPUSH friends "Jeff"
```

Wir können auch mehrere Elemente in einem mal hinzufügen:

```
RPUSH friends "Alice" "Bob" "Jeff"
```

Subset

Mit dem `LRANGE` Kommando bekommen wir ein Subset der Liste: `LRANGE <list> <startindex> <endindex>`

```
LRANGE friends 0 -1
```

-1: Bis zum Ende -2: Bis Vorletze etc...

Länge

`LLEN <list>` gibt uns die Länge der Liste zurück.

Löschen

Mit , `LPOP <list>`, `RPOP <list>`. Können wir das erste oder das letzte Element einer Liste löschen. Das gelöschte Element wird returned.

Sets

Sets sind eine unsortierte Sammlung von Strings. Im Gegensatz zu Listen, sind Sie nicht sortiert und der Zugriff erfolgt deshalb nicht über einen Index, sondern über den String-Wert selber.

Add

Mit `SADD <set> <value>` können wir einen String mit dem Wert `value` dem set hinzufügen. Wie bei Lists muss ein Set nicht initialisiert werden. Returnd `0`, wenn der String bereits im Set vorhanden ist.

```
SADD superpowers "flight"
SADD superpowers "x-ray vision" "reflexes"
```

Remove

Mit `SREM <set> <value>` können wir einen String wieder aus dem Set entfernen. Falls etwas gelöscht wurde, returnt diese Funktion `1`.

```
SREM superpowers "reflexes"
```

Sorted Sets

Sorted Sets sind den normalen Sets sehr ähnlich. Sie enthalten eine Sammlung aus eindeutigen Strings. Wie der Name `sorted` schon sagt, sind sie aber, im Gegensatz zum normalen Set, sortiert. Die Sortierung findet über einen `score` statt. Es wird vom kleinsten `score` zum größten `score` sortiert.
Die `score`s dürfen sich wiederholen.

ADD

Mit `ZADD <set-name> <score> <value>` können wir ein neues Element in unser sorted set einfügen.

```
ZADD hackers 1940 "Alan Kay"
ZADD hackers 1906 "Grace Hopper"
ZADD hackers 1953 "Richard Stallman"
ZADD hackers 1965 "Yukihiro Matsumoto"
ZADD hackers 1916 "Claude Shannon"
ZADD hackers 1969 "Linus Torvalds"
ZADD hackers 1957 "Sophie Wilson"
ZADD hackers 1912 "Alan Turing"
```

ZRANGE

Mit `ZRANGE <startindex> <endindex>` können wir analog zu den Listen ein oder mehrere Elemente aus unserem Sorted Set bekommen.

```
ZRANGE hackers 0 -1
```

Hashes

Werden genutzt um einem String-Feld einen Wert zuzuweisen. Sie eignen sich super um Objekte zu repräsentieren.
Mit `HSET <hash-name> <field> <value>` können wir ein Feld - Wert Paar in einem Hash anlegen.
Auch hier gilt: Ein hash muss nicht erst initialisiert werden. Wir können einfach in ein noch nicht vorhandenden Hash Werte ablegen und Redis legt einen neuen Hash an.

```
HSET user:1000 name "John Smith"
HSET user:1000 email "email@email.com"
HSET user:1000 password "s3cr3t"
```

Um alle Felder eines Hashes zu bekommen nutzen wir `HGETALL <hash>`

```
HGETALL user:1000
```

Set multiple

Wir können auch mehrere Felder in einem Kommando anlegen. Dazu nutzen wir `HMSET <hash> [... <field>, <value>]`.

```
HMSET user:1001 name "Mary Jones" password "hidden" email "mjones@example.com"
```

Get single field

Wenn uns nur ein einziges Feld aus einem Hash interessiert bekommen wir dieses über `HGET <hash> <field>`

```
HGET user:1001 name
```

Increment

Auch Hashes bieten Funktionen für Increment und Decrement.
`HINCRBY <hash> <field> <amount>` und `HDECRBY <hash> <field> <amount>`. Sie funktionieren analog zu den `DECRBY` und `INCRBY` Funktionen der normalen Key-Value Paaren. Auch sie bieten Sicherheit bei gleichzeitigem Zugriff auf die gleichen Daten.

```
HSET user:1000 visits 10
```

```
HINCRBY user:1000 visits 1
```

Anwendungsfälle:

Meistens in moderner micro-services Architektur.

REDIS Strings:

- Session Cache
- Queues
- Usage & Meterd Billing

REDIS Lists:

- Social Networking Sites (Twitter timelines, homepage feeds, trending feeds)
- RSS Feeds
- Leaderboards

REDIS Sets:

- Analyzing Ecommerce Sales (Customer behavior)
- IP Address Tracking
- Inappropriate Content Filtering

REDIS Sorted Sets:

- Q&A Platforms
- Scoreboards
- Task Scheduling Service
- Geo Hashing

REDIS Hashes:

- User Profiles
- User Posts
- Metrics

Data Persitance

Default - Snapshots

Standardmäßig nutzt Redis `Snapshots` um Daten zu speichern. Die Daten werden in eine Datei `dump.rdb` gespeichert. Mit `save <seconds> <changes>` kann eingestellt werden, nach wievielen Sekunden oder wievielen Änderungen am Datensatz Redis einen neuen Snapshot erstellen soll. Sonst kann mit `SAVE` oder `BGSAVE` manuell gespeichert werden.

Append only file

Eine andere Methode ist das `Append-only-file`. Das bietet mehr Sicherheit gegen Datenverlust (Falls zb. ein Stromausfall vor einem Snapshot aber nach Datenänderungen eintritt), da es nach jeder Datensatzänderung

diese auf die Festplatte schreibt. Diesen Modus kann man mit `appendonly yes` einschalten. Wenn man nun die Instanz neustartet, dann führt Redis alle mitgeschriebenen Kommandos beim Start aus, sodass der Datensatz auf dem alten Stand wieder ist.

Wenn wir also 100 mal einen Wert erhöhen (`INCR <key>`) wird dies 100 mal in das `AOF` geschrieben. Dadurch wird das `AOF` sehr schnell sehr groß. Aus diesem Grund kann Redis im Hintergrund das `AOF` neubauen. Dies kann man mit `BGREWRITEAOF` einmalig machen. Unter Redis 2.4 und neuer kann man dies auch automatisch machen lassen. Dazu muss dann die Konfiguration entsprechend angepasst werden.

Mehr zum Thema

Persitance Docs