Basic Redis Commands

We communicate with the Redis server by using commands to perform operations on the database.

A command has one or more arguments like keys and values that help us perform these operations.

In this lesson, we cover only common and basic commands.

The KEYS command

The KEYS command takes in a pattern as an argument and returns all the keys that matches that pattern.

Syntax:

```
KEYS pattern
```

To return a list of all the keys in the database, we use the `*` (asterisk) symbol.

Example:

```
KEYS *
```

If there are no keys present, it will return one of the following messages.

Output:

```
(empty list or set)

(empty array)
```

The SET & GET commands

As mentioned before, Redis is a key:value store. There are no rows and columns like in a relational database, only keys and their corresponding values.

We can set a key:value pair by using the `SET` command. This command takes the key name and its corresponding value as arguments.

Syntax:

```
SET name value
```

Example:

```
SET fName John
```

In the example above we create a new key with the name **fName** and the value *John*.

We can get the value of a key by using the `GET` command. This command takes the key name as an argument.

Syntax:

```
GET name
```

Example:

```
GET fName
```

In the example above we access the value of **fName**, which is John.

Output:

```
"John"
```

The `SET` command is also used to update the value of a key.

Example:

```
SET fName Jane
```

If we access the key now, it will show the new value.

Example:

```
GET fName
```

Output:

```
"Jane"
```

Sometimes, this behavior isn't what we want. Redis allows us to set a value only if the key doesn't exist yet with the `SETNX` command.

Example:

```
SETNX fName John
```

If we try to change the **fName** key with this command, it won't work because it already exists.

Output:

```
(integer) 0
```

The `SETNX` command allows us to work a little more defensively, we won't be able to overwrite any data accidentally.

The APPEND command

Redis allows us to append data to the end of values with the `APPEND` command. This command takes the key name and the value to append as arguments.

Syntax:

```
APPEND name value
```

Example:

```
APPEND fName " Doe"
```

We wrap the value in quotes because we want a space to separate the name and surname.

If we check the value of the key now, we can see that it appended the surname.

Example:

```
GET fName
```

Output:

```
"John Doe"
```

The DEL command

If we want to delete a key and its value, we can use the `DEL` command. This command takes the key name as an argument.

Syntax:

```
DEL name
```

Example:

```
DEL fName
```

In the example above we delete the **fName** key from the database.

The output message is a number that shows how many records were affected.

Output:

```
(integer) 1
```

This means we successfully deleted the key and its value. We can test it by running the `GET` command.

Example:

```
GET fName
```

Output:

```
(nil)
```

This time the output message shows **nil**, which means the key does not exist.

If we pass in a key that doesn't exist, no keys will be deleted.

Example:

```
DEL someKey
```

Because the key doesn't exist, Redis can't update any records and the output message will show 0.

Output:

```
(integer) 0
```

The asterisk * pattern

As mentioned before, the `*` symbol will show a list of all the keys in the database.

To demonstrate, let's add multiple keys to the database.

Example:

```
SET fname John
SET lname Doe
SET age 31
```

Now when we use the `*` as a pattern for the `KEYS` command, it will show this list of keys.

Example:

```
KEYS *
```

Output:

```
1) "lname"
2) "fname"
3) "age"
```

The asterisk symbol represents *anything*. We can use it with more patterns to help us refine our search for keys.

For example, if we add `*` after the word 'age', it will show all the keys that start with the word 'age'.

Example:

```
KEYS age*
```

Output:

```
1) "age"
```

It doesn't have to be a full word, it can be part of a word or even a single letter.

Example:

```
KEYS a*
```

Output:

```
1) "age"
```

The command reads: `a + anything after it`

When we add a `*` before a word or letter, it will show all the keys that end with that word or letter.

Example:

```
KEYS *e
```

Output:

```
1) "lname"
2) "fname"
3) "age"
```

Because all of our key names end with an e, it will return all of them.

The command reads: `anything that ends with e`

When we add a `*` before and after the word or letter, it will show all the keys that contain that word or letter.

Example:

```
KEYS *g*
```

Output:

```
1) "age"
```

The command reads: `anything with g inside it`

Example:

```
KEYS *name*
```

Output:

```
1) "lname"
2) "fname"
```

The command reads: `anything with name inside it`

The FLUSHALL command

The `FLUSHALL` command is used to clear all the databases on the server, removing all keys and values.

Example:

```
FLUSHALL
```

To demonstrate that the database was cleared, we can use the `KEYS` command.

Example:

```
KEYS *
```

Output:

```
(empty array)
```

The `FLUSHALL` command returns only after the database is fully cleared. Depending on the size of the database, this may take some time, which is why we can let it return as soon as the command is received by the server with `ASYNC` .

Syntax:

```
FLUSHALL ASYNC
```

Key expiration

We can set an expiration time on keys so that after a certain amount of time, they're cleared from the database automatically.

As an example, let's consider that we need to store session IDs that become invalid after a certain period of time. If we set an expiration time on that key, it will automatically delete the key when the time has passed.

To see if a key has an expiration on it, and when that time expires in seconds, we can use the `TTL` (Time To Live) command.

Syntax:

```
TTL name
```

Example:

```
SET num 8

TTL num
```

In the example above we add a new key called **num** with a value of *8*, then check its expiration with the `TTL` command.

Output:

```
(integer) -1
```

The command returns a value of -1, which means that the key has no expiration time.

We can set an expiration on a key by using the `EXPIRE` command. This command takes the key name and time to expire in seconds as arguments.

Syntax:

```
EXPIRE name seconds
```

Example:

```
EXPIRE num 30
```

In the example above we specify that we want the **num** key to live for *30* seconds.

Output:

```
(integer) 1
```

The output returns success. When we check the `TTL` again now, it will show how many seconds are left.

Once the key expires, the output will be different.

Output:

```
(integer) -2
```

It's important to remember that -1 means an expiration has not been set, whereas -2 means that a key with an expiration has reached the expiration time.

We can add an expiration time directly when creating the key with the `SETEX` command. This command accepts the name of the key, expiration in seconds, and then the value as arguments.

Syntax:

```
SETEX name seconds value
```

Example:

```
SETEX fName 30 John
```

The EXISTS command

We can check if a key exists with the `EXISTS` command. This command takes the key name as an argument.

Syntax:

```
EXISTS name
```

Example:

```
EXISTS fName
```

Because the key was set with `SETEX` in the previous section, it expired and no longer exists.

Output:

```
(integer) 0
```

If we `SET` the key and check if it exists again, it will show 1 in the output.

Example:

```
SET fName John

EXISTS fName
```

Output:

```
(integer) 1
```

The STRLEN command

We can check the length of the value stored in a key with the `STRLEN` command. This command takes the key name as an argument.

Syntax:

```
STRLEN name
```

Example:

```
STRLEN fName
```

Output:

```
(integer) 4
```

The output shows 4 because there are four characters in the name John.

If we do the same with a number, it will show the amount of digits in the number.

Example:

```
SET num 30

STRLEN num
```

Output:

```
(integer) 2
```

The output shows 2 because there are two digits in the number 30.

The DUMP & RESTORE commands

We can serialize the value of a key in a Redis specific format by using the `DUMP` command. This command takes the key name as an argument.

Syntax:

```
DUMP name
```

Example:

```
SET fullName "John Doe"

DUMP fullName
```

Output:

```
"\x00\bJohn Doe\t\x00\x98\xf3~\x15\xa8\xc7\x8d}"
```

The serialized value can be restored in this or another database with the `RESTORE` command. This command takes the new key name, followed by a `TTL` value and the serialized value as arguments.

Syntax:

```
RESTORE name ttl serializedValue
```

If we specify 0 as the TTL expiration, the key will not have any expiration set.

If the key already exists, we can add the `REPLACE` modifier to command to replace the value in the key.

Syntax:

```
RESTORE name ttl serializedValue REPLACE
```

Example:

```
DEL fullName

RESTORE name 0 "\x00\bJohn Doe\t\x00\x98\xf3~\x15\xa8\xc7\x8d}"
```

In the example above we first delete old **fullName** key, then restore the serialized one to a new key called **name** with no expiry.

The HELP command

Some commands allow us to get more information about their subcommands by using the `HELP` subcommand.

Syntax:

```
COMMAND HELP
```

Example:

```
CLIENT HELP
```

The example above will return the subcommands of the CLIENT command.

Output:

```
1) CLIENT <subcommand> arg arg ... arg. Subcommands are:
2) ID                    -- Return the ID of the current connection.
3) GETNAME               -- Return the name of the current connection.
4) KILL <ip:port>        -- Kill connection made from <ip:port>.
5) KILL <option> <value> [option value ...] -- Kill connections. Options are:
6)      ADDR <ip:port>                      -- Kill connection made from <ip:port>
7)      TYPE (normal|master|replica|pubsub) -- Kill connections by type.
8)      USER <username>   -- Kill connections authenticated with such user.
9)      SKIPME (yes|no)   -- Skip killing current connection (default: yes).
10) LIST [options ...]     -- Return information about client connections. Options:
11)      TYPE (normal|master|replica|pubsub) -- Return clients of specified type.
12) PAUSE <timeout>        -- Suspend all Redis clients for <timout> milliseconds.
13) REPLY (on|off|skip)    -- Control the replies sent to the current connection.
14) SETNAME <name>         -- Assign the name <name> to the current connection.
15) UNBLOCK <clientid> [TIMEOUT|ERROR] -- Unblock the specified blocked client.
16) TRACKING (on|off) [REDIRECT <id>] [BCAST] [PREFIX first] [PREFIX second] [OPTIN]
17) GETREDIR               -- Return the client ID we are redirecting to when trackin
```