Redis Keys and Values

**Every data object that you store in a Redis database has its own unique key.**

The key is a string that you pass to Redis commands to retrieve the corresponding object or modify its data.

The data object associated with a particular key is known as the value and the two together are known as  key-value pair.

# Content of Keys

**A key** is **typically** a textual name that **has some meaning within your data model.**

Unlike variable names in a programming language, Redis keys have few restrictions on their format, so keys with whitespace or punctuation characters are mostly fine

(for example, "1st Attempt", or "% of price in $").

Redis doesn't support namespaces or other categories for keys, so **you must take care to avoid name collisions.**

However, **there** **is a convention for using the colon ":" character to split keys into sections** (for example, "person:1", "person:2", "office:London", "office:NewYork:1"). You can use this as a simple way to collect keys together into categories.

Although keys are usually textual, Redis actually implements *binary-safe* keys, so **you can use any sequence of bytes as a valid key**, **such as a JPEG file or a struct value from your app**. **The empty string is also a valid key in Redis.**

I add: *a struct value is probably a composition of values of fields in something like value objects.*

From Wikipedia:(Not clear at the moment)

*A binary-safe function is one that treats its input as a raw stream of bytes and ignores every textual aspect it may have*

There are also a few other things to bear in mind about keys:

* Very long keys are not a good idea. For instance, a key of 1024 bytes is a bad idea not only memory-wise, but also because the lookup of the key in the dataset may require several costly key-comparisons. Even when the task at hand is to match the existence of a large value, hashing it (for example with SHA1) is a better idea, especially from the perspective of memory and bandwidth.

I add:

*I assume it means that when facing a hash collision, you will have to compare several keys where each one is 1KB of data and this comparison could be costly.*

*By hashing the keys, ( It is highly unlikely for two different inputs to produce the same output hash, and The same input will always produce the same output) you will have a shorter key representing the original one.*

*Also, when sending kilobytes of data to Redis just for the keys, your bandwidth is also going to get hit.*

* Very short keys are often not a good idea. There is little point in writing "u1000flw" as a key if you can instead write "user:1000:followers". The latter is more readable and the added space is minor compared to the space used by the key object itself and the value object. While short keys will obviously consume a bit less memory, your job is to find the right balance.
* Try to stick with a schema. For instance, "object-type:id" is a good idea, as in "user:1000". Dots or dashes are often used for multi-word fields, as in "comment:4321:reply.to" or "comment:4321:reply-to".
* The maximum allowed key size is 512 MB.

## Hash Tags

Redis uses hashing to retrieve the value associated with a key in a highly efficient way. Hashing involves combining the raw byte values from the key to produce an integer index number. The index is then used to locate the hash slot where the value for the key is stored.

**Normally, the whole key is used to calculate the hash index, but there are some situations where you need to hash only a part of the key**. You can select the section of the key you want to hash using a pair of curly braces {...} to create a hashtag.

For example, the keys person:1 and person:2 produce different hash indices but {person}:1 and {person}:2 produce the same index because only the person hashtag section in the braces is used for the hash calculation.

Skipped this for now:

A common use of hashtags is to allow multi-key operations with a clustered database. Redis doesn't allow most multi-key operations in a clustered database unless all the keys produce the same hash index.

For example, the SINTER command finds the intersection of two different set values. This means that the command

SINTER group:1 group:2

won't work with a clustered database but

SINTER **{**group**}**:1 **{**group**}**:2

will work because the hashtag ensures the two keys produce the same hash index.

Note that although hashtags are useful in certain cases, you shouldn't make a habit of using them generally. If you have too many keys mapped to the same hash slot then this will eventually harm the performance of your database. See [Database clustering](https://redis.io/docs/latest/operate/rs/databases/durability-ha/clustering/) for more information about how to use hashtags:

<https://redis.io/docs/latest/operate/rs/databases/durability-ha/clustering/>

# Altering and querying the key space

There are commands that are not defined on particular types, but **are useful in order to interact with the space of keys,** and thus, can be **used with keys of any type.**

For example the [EXISTS](https://redis.io/docs/latest/commands/exists/) command returns 1 or 0 to signal if a given key exists or not in the database, while the [DEL](https://redis.io/docs/latest/commands/del/) command **deletes a key and associated value, whatever the value is** and returns 1 or 0 depending on whether the key was removed (it existed) or not (there was no such key with that name)

the [TYPE](https://redis.io/docs/latest/commands/type/) command returns the kind of value stored at the specified key:

> set mykey x

OK

> type mykey

string

> del mykey

(integer) 1

> type mykey

Non

# Key Expiration

<https://redis.io/docs/latest/commands/expire/>