Tutorial: Design and implementation of a simple Twitter clone using PHP and the Redis key-value store

This article describes the **design and implementation of a** very **simple Twitter** clone written using PHP with **Redis** as the only database. The programming community has traditionally considered key-value stores as a special purpose database that couldn't be used as a drop-in replacement for a relational database for the development of web applications. This article will try to show that **Redis** data structures on top of a key-value layer are an effective data model to implement many kinds of applications.

Before continuing, you may want to spend a few seconds playing with the Retwis online demo, to check what we are going to actually model. Long story short: it is a toy, but complex enough to be a foundation in order to learn how to create more complex applications.

Note: the original version of this article was written in 2009 when **Redis** was released. It was not exactly clear at that time that the **Redis** data model was suitable to write entire applications. Now after 5 years there are many cases of applications using **Redis** as their main store, so the goal of the article today is to be a tutorial for **Redis** newcomers. You'll learn how to design a **simple** data layout using **Redis**, and how to apply different data structures.

Our **Twitter** clone, called Retwis, is structurally **simple**, has very good performance, and can be distributed among any number of web and **Redis** servers with little efforts. You can find the source code here.

I used PHP for the example since it can be read by everybody. The same (or better) results can be obtained using Ruby, Python, Erlang, and so on. A few clones exist (however not all the

clones use the same data layout as the current version of this tutorial, so please, stick with the official **PHP** implementation for the sake of following the article better).

- Retwis-RB is a port of Retwis to Ruby and Sinatra written by Daniel Lucraft! Full source code is included of course, and a link to its Git repository appears in the footer of this article. The rest of this article targets **PHP**, but Ruby programmers can also check the Retwis-RB source code since it's conceptually very similar.
- Retwis-J is a port of Retwis to Java, using the Spring Data Framework, written by Costin Leau. Its source code can be found on GitHub, and there is comprehensive documentation available at springsource.org.

What is a key-value store?

The essence of a key-value store is the ability to store some data, called a *value*, inside a key. The value can be retrieved later only if we know the specific key it was stored in. There is no direct way to search for a key by value. In some sense, it is like a very large hash/dictionary, but it is persistent, i.e. when your application ends, the data doesn't go away. So, for example, I can use the command SET to store the value *bar* in the key *foo*:

```
SET foo bar
```

Redis stores data permanently, so if I later ask "*What is the value stored in key foo?*" **Redis** will reply with *bar*:

```
GET foo => bar
```

Other common operations provided by key-value stores are DEL, to delete a given key and its associated value, SET-if-not-exists (called SETNX on **Redis**), to assign a value to a key only if the key does not already exist, and INCR, to atomically increment a number stored in a given key:

```
SET foo 10
INCR foo => 11
INCR foo => 12
INCR foo => 13
```

Atomic operations

There is something special about INCR. You may wonder why **Redis** provides such an operation if we can do it ourselves with a bit of code? After all, it is as **simple** as:

```
x = GET foo
x = x + 1
SET foo x
```

The problem is that incrementing this way will work as long as there is only one client working with the key *foo* at one time. See what happens if two clients are accessing this key at the same time:

```
x = GET foo (yields 10)
y = GET foo (yields 10)
x = x + 1 (x is now 11)
y = y + 1 (y is now 11)
SET foo x (foo is now 11)
SET foo y (foo is now 11)
```

Something is wrong! We incremented the value two times, but instead of going from 10 to 12, our key holds 11. This is because the increment done with `GET / increment / SET` *is not an atomic operation*. Instead the INCR provided by **Redis**, Memcached, ..., are atomic implementations, and the server will take care of protecting the key during the time needed to complete the increment in order to prevent simultaneous accesses.

What makes **Redis** different from other key-value stores is that it provides other operations similar to INCR that can be used to model complex problems. This is why you can use **Redis** to write whole web applications without using another database like an SQL database, and without going crazy.

Beyond key-value stores: lists

In this section we will see which **Redis** features we need to build our **Twitter** clone. The first thing to know is that **Redis** values can be more than strings. **Redis** supports Lists, Sets, Hashes, Sorted Sets, Bitmaps, and HyperLogLog types as values, and there are atomic operations to operate on them so we are safe even with multiple accesses to the same key. Let's start with Lists:

```
LPUSH mylist a (now mylist holds 'a')
LPUSH mylist b (now mylist holds 'b','a')
LPUSH mylist c (now mylist holds 'c','b','a')
```

LPUSH means *Left Push*, that is, add an element to the left (or to the head) of the list stored in *mylist*. If the key *mylist* does not exist it is automatically created as an empty list before the PUSH operation. As you can imagine, there is also an RPUSH operation that adds the element to the right of the list (on the tail). This is very useful for our **Twitter** clone. User updates can be added to a list stored in `username:updates`, for instance.

There are operations to get data from Lists, of course. For instance, LRANGE returns a range from the list, or the whole list.

```
LRANGE mylist 0 1 => c,b
```

LRANGE uses zero-based indexes - that is the first element is 0, the second 1, and so on. The command arguments are `LRANGE key first-index last-index`. The *last-index* argument can be negative, with a special meaning: -1 is the last element of the list, -2 the penultimate, and so on. So, to get the whole list use:

```
LRANGE mylist 0 -1 => c,b,a
```

Other important operations are LLEN that returns the number of elements in the list, and LTRIM that is like LRANGE but instead of returning the specified range *trims* the list, so it is like *Get range from mylist, Set this range as new value* but does so atomically.

The Set data type

Currently we don't use the Set type in this tutorial, but since we use Sorted Sets, which are kind of a more capable version of Sets, it is better to start introducing Sets first (which are a very useful data structure per se), and later Sorted Sets.

There are more data types than just Lists. **Redis** also supports Sets, which are unsorted collections of elements. It is possible to add, remove, and test for existence of members, and perform the intersection between different Sets. Of course it is possible to get the elements of a Set. Some examples will make it more clear. Keep in mind that SADD is the *add to set*

operation, SREM is the *remove from set* operation, *sismember* is the *test if member* operation, and SINTER is the *perform intersection* operation. Other operations are SCARD to get the cardinality (the number of elements) of a Set, and SMEMBERS to return all the members of a Set.

```
SADD myset a
SADD myset b
SADD myset foo
SADD myset bar
SCARD myset => 4
SMEMBERS myset => bar,a,foo,b
```

Note that SMEMBERS does not return the elements in the same order we added them since Sets are *unsorted* collections of elements. When you want to store in order it is better to use Lists instead. Some more operations against Sets:

```
SADD mynewset b
SADD mynewset foo
SADD mynewset hello
SINTER myset mynewset => foo,b
```

SINTER can return the intersection between Sets but it is not limited to two Sets. You may ask for the intersection of 4,5, or 10000 Sets. Finally let's check how SISMEMBER works:

```
SISMEMBER myset foo => 1
SISMEMBER myset notamember => 0
```

The Sorted Set data type

Sorted Sets are similar to Sets: collection of elements. However in Sorted Sets each element is associated with a floating point value, called the *element score*. Because of the score, elements inside a Sorted Set are ordered, since we can always compare two elements by score (and if the score happens to be the same, we compare the two elements as strings).

Like Sets in Sorted Sets it is not possible to add repeated elements, every element is unique. However it is possible to update an element's score.

Sorted Set commands are prefixed with Z. The following is an example of Sorted Sets usage:

```
ZADD zset 10 a
ZADD zset 5 b
ZADD zset 12.55 c
ZRANGE zset 0 -1 => b,a,c
```

In the above example we added a few elements with ZADD, and later retrieved the elements with ZRANGE. As you can see the elements are returned in order according to their score. In order to check if a given element exists, and also to retrieve its score if it exists, we use the ZSCORE command:

```
ZSCORE zset a => 10
ZSCORE zset non_existing_element => NULL
```

Sorted Sets are a very powerful data structure, you can query elements by score range, lexicographically, in reverse order, and so forth. To know more please check the Sorted Set sections in the official **Redis** commands documentation.

The Hash data type

This is the last data structure we use in our program, and is extremely easy to gasp since there is an equivalent in almost every programming language out there: Hashes. **Redis** Hashes are basically like Ruby or Python hashes, a collection of fields associated with values:

```
HMSET myuser name Salvatore surname Sanfilippo country Italy
HGET myuser surname => Sanfilippo
```

HMSET can be used to set fields in the hash, that can be retrieved with HGET later. It is possible to check if a field exists with HEXISTS, or to increment a hash field with HINCRBY and so forth.

Hashes are the ideal data structure to represent *objects*. For example we use Hashes in order to represent Users and Updates in our **Twitter** clone.

Okay, we just exposed the basics of the **Redis** main data structures, we are ready to start coding!

Prerequisites

If you haven't downloaded the Retwis source code already please grab it now. It contains a few **PHP** files, and also a copy of Predis, the **PHP** client library we use in this example.

Another thing you probably want is a working **Redis** server. Just get the source, build with `make`, run with `./redis-server`, and you're ready to go. No configuration is required at all in order to play with or run Retwis on your computer.

Data layout

When working with a relational database, a database schema must be designed so that we'd know the tables, indexes, and so on that the database will contain. We don't have tables in **Redis**, so what do we need to design? We need to identify what keys are needed to represent our objects and what kind of values this keys need to hold.

Let's start with Users. We need to represent users, of course, with their username, userid, password, the set of users following a given user, the set of users a given user follows, and so on. The first question is, how should we identify a user? Like in a relational DB, a good solution is to identify different users with different numbers, so we can associate a unique ID with every user. Every other reference to this user will be done by id. Creating unique IDs is very **simple** to do by using our atomic INCR operation. When we create a new user we can do something like this, assuming the user is called "antirez":

```
INCR next_user_id => 1000
HMSET user:1000 username antirez password p1pp0
```

Note: you should use a hashed password in a real application, for simplicity we store the password in clear text.

We use the `next_user_id` key in order to always get a unique ID for every new user. Then we use this unique ID to name the key holding a Hash with user's data. *This is a common design pattern* with key-values stores! Keep it in mind. Besides the fields already defined, we need some more stuff in order to fully define a User. For example, sometimes it can be useful to be able to get the user ID from the username, so every time we add a user, we also populate the `users` key, which is a Hash, with the username as field, and its ID as value.

```
HSET users antirez 1000
```

This may appear strange at first, but remember that we are only able to access data in a direct way, without secondary indexes. It's not possible to tell **Redis** to return the key that holds a specific value. This is also *our strength*. This new paradigm is forcing us to organize data so that everything is accessible by *primary key*, speaking in relational DB terms.

Followers, following, and updates

There is another central need in our system. A user might have users who follow them, which we'll call their followers. A user might follow other users, which we'll call a following. We have a perfect data structure for this. That is... Sets. The uniqueness of Sets elements, and the fact we can test in constant time for existence, are two interesting features. However what about also remembering the time at which a given user started following another one? In an enhanced version of our **simple Twitter** clone this may be useful, so instead of using a **simple** Set, we use a Sorted Set, using the user ID of the following or follower user as element, and the unix time at which the relation between the users was created, as our score.

So let's define our keys:

```
followers:1000 => Sorted Set of uids of all the followers users
following:1000 => Sorted Set of uids of all the following users
```

We can add new followers with:

```
ZADD followers:1000 1401267618 1234 => Add user 1234 with time 1401267618
```

Another important thing we need is a place were we can add the updates to display in the user's home page. We'll need to access this data in chronological order later, from the most recent update to the oldest, so the perfect kind of data structure for this is a List. Basically every new update will be LPUSHed in the user updates key, and thanks to LRANGE, we can implement pagination and so on. Note that we use the words *updates* and *posts* interchangeably, since updates are actually "little posts" in some way.

```
posts:1000 => a List of post ids - every new post is LPUSHed here.
```

This list is basically the User timeline. We'll push the IDs of her/his own posts, and, the IDs of all the posts of created by the following users. Basically, we'll implement a write fanout.

Authentication

OK, we have more or less everything about the user except for authentication. We'll handle authentication in a **simple** but robust way: we don't want to use **PHP** sessions, as our system must be ready to be distributed among different web servers easily, so we'll keep the whole state in our **Redis** database. All we need is a random **unguessable** string to set as the cookie of an authenticated user, and a key that will contain the user ID of the client holding the string.

We need two things in order to make this thing work in a robust way. First: the current authentication *secret* (the random unguessable string) should be part of the User object, so when the user is created we also set an `auth` field in its Hash:

```
HSET user:1000 auth fea5e81ac8ca77622bed1c2132a021f9
```

Moreover, we need a way to map authentication secrets to user IDs, so we also take an `auths` key, which has as value a Hash type mapping authentication secrets to user IDs.

```
HSET auths fea5e81ac8ca77622bed1c2132a021f9 1000
```

In order to authenticate a user we'll do these **simple** steps (see the `login.php` file in the Retwis source code):

- Get the username and password via the login form.
- Check if the `username` field actually exists in the `users` Hash.
- If it exists we have the user id, (i.e. 1000).
- Check if user:1000 password matches, if not, return an error message.
- Ok authenticated! Set "fea5e81ac8ca77622bed1c2132a021f9" (the value of user:1000 `auth` field) as the "auth" cookie.

This is the actual code:

```
include("retwis.php");

# Form sanity checks
if (!gt("username") || !gt("password"))
    goback("You need to enter both username and password to login.");

# The form is ok, check if the username is available
$username = gt("username");
$password = gt("password");
$r = redisLink();
$userid = $r->hget("users",$username);
if (!$userid)
    goback("Wrong username or password");
$realpassword = $r->hget("user:$userid","password");
if ($realpassword != $password)
    goback("Wrong username or password");

# Username / password OK, set the cookie and redirect to index.php
$authsecret = $r->hget("user:$userid","auth");
setcookie("auth",$authsecret,time()+3600*24*365);
header("Location: index.php");
```

This happens every time a user logs in, but we also need a function `isLoggedIn` in order to check if a given user is already authenticated or not. These are the logical steps preformed by the `isLoggedIn` function:

- Get the "auth" cookie from the user. If there is no cookie, the user is not logged in, of course. Let's call the value of the cookie `<authcookie>`.
- Check if `<authcookie>` field in the `auths` Hash exists, and what the value (the user ID) is (1000 in the example).
- In order for the system to be more robust, also verify that user:1000 auth field also matches.
- OK the user is authenticated, and we loaded a bit of information in the $User global variable.

The code is simpler than the description, possibly:

```
function isLoggedIn() {
    global $User, $_COOKIE;

    if (isset($User)) return true;

    if (isset($_COOKIE['auth'])) {
        $r = redisLink();
        $authcookie = $_COOKIE['auth'];
        if ($userid = $r->hget("auths",$authcookie)) {
            if ($r->hget("user:$userid","auth") != $authcookie) return false;
            loadUserInfo($userid);
            return true;
        }
    }
    return false;
}

function loadUserInfo($userid) {
    global $User;

    $r = redisLink();
    $User['id'] = $userid;
    $User['username'] = $r->hget("user:$userid","username");
    return true;
}
```

Having `loadUserInfo` as a separate function is overkill for our application, but it's a good approach in a complex application. The only thing that's missing from all the authentication is the logout. What do we do on logout? That's **simple**, we'll just change the random string in user:1000 `auth` field, remove the old authentication secret from the `auths` Hash, and add the new one.

Important: the logout procedure explains why we don't just authenticate the user after looking up the authentication secret in the `auths` Hash, but double check it against user:1000 `auth` field. The true authentication string is the latter, while the `auths` Hash is just an authentication field that may even be volatile, or, if there are bugs in the program or a script

gets interrupted, we may even end with multiple entries in the `auths` key pointing to the same user ID. The logout code is the following (`logout.php`):

```
include("retwis.php");

if (!isLoggedIn()) {
    header("Location: index.php");
    exit;
}

$r = redisLink();
$newauthsecret = getrand();
$userid = $User['id'];
$oldauthsecret = $r->hget("user:$userid","auth");

$r->hset("user:$userid","auth",$newauthsecret);
$r->hset("auths",$newauthsecret,$userid);
$r->hdel("auths",$oldauthsecret);

header("Location: index.php");
```

That is just what we described and should be **simple** to understand.

Updates

Updates, also known as posts, are even simpler. In order to create a new post in the database we do something like this:

```
INCR next_post_id => 10343
HMSET post:10343 user_id $owner_id time $time body "I'm having fun with Retwis
```

As you can see each post is just represented by a Hash with three fields. The ID of the user owning the post, the time at which the post was published, and finally, the body of the post, which is, the actual status message.

After we create a post and we obtain the post ID, we need to LPUSH the ID in the timeline of every user that is following the author of the post, and of course in the list of posts of the

author itself (everybody is virtually following herself/himself). This is the file `post.php` that shows how this is performed:

```
include("retwis.php");

if (!isLoggedIn() || !gt("status")) {
    header("Location:index.php");
    exit;
}

$r = redisLink();
$postid = $r->incr("next_post_id");
$status = str_replace("\n"," ",gt("status"));
$r->hmset("post:$postid","user_id",$User['id'],"time",time(),"body",$status);
$followers = $r->zrange("followers:".$User['id'],0,-1);
$followers[] = $User['id']; /* Add the post to our own posts too */

foreach($followers as $fid) {
    $r->lpush("posts:$fid",$postid);
}
# Push the post on the timeline, and trim the timeline to the
# newest 1000 elements.
$r->lpush("timeline",$postid);
$r->ltrim("timeline",0,1000);

header("Location: index.php");
```

The core of the function is the `foreach` loop. We use ZRANGE to get all the followers of the current user, then the loop will LPUSH the push the post in every follower timeline List.

Note that we also maintain a global timeline for all the posts, so that in the Retwis home page we can show everybody's updates easily. This requires just doing an LPUSH to the `timeline` List. Let's face it, aren't you starting to think it was a bit strange to have to sort things added in chronological order using `ORDER BY` with SQL? I think so.

There is an interesting thing to notice in the code above: we used a new command called LTRIM after we perform the LPUSH operation in the global timeline. This is used in order to

trim the list to just 1000 elements. The global timeline is actually only used in order to show a few posts in the home page, there is no need to have the full history of all the posts.

Basically LTRIM + LPUSH is a way to create a *capped collection* in **Redis**.

Paginating updates

Now it should be pretty clear how we can use LRANGE in order to get ranges of posts, and render these posts on the screen. The code is **simple**:

```
function showPost($id) {
    $r = redisLink();
    $post = $r->hgetall("post:$id");
    if (empty($post)) return false;

    $userid = $post['user_id'];
    $username = $r->hget("user:$userid","username");
    $elapsed = strElapsed($post['time']);
    $userlink = "<a class=\"username\" href=\"profile.php?u=".urlencode($userr

    echo('<div class="post">'.$userlink.' '.utf8entities($post['body'])."<br>"
    echo('<i>posted '.$elapsed.' ago via web</i></div>');
    return true;
}

function showUserPosts($userid,$start,$count) {
    $r = redisLink();
    $key = ($userid == -1) ? "timeline" : "posts:$userid";
    $posts = $r->lrange($key,$start,$start+$count);
    $c = 0;
    foreach($posts as $p) {
        if (showPost($p)) $c++;
        if ($c == $count) break;
    }
    return count($posts) == $count+1;
}
```

`showPost` will simply convert and print a Post in HTML while `showUserPosts` gets a range of posts and then passes them to `showPosts`.

*Note: LRANGE is not very efficient if the list of posts start to be very big, and we want to access elements which are in the middle of the list, since **Redis** Lists are backed by linked lists. If a system is designed for deep pagination of million of items, it is better to resort to Sorted Sets instead.*

Following users

It is not hard, but we did not yet check how we create following / follower relationships. If user ID 1000 (antirez) wants to follow user ID 5000 (pippo), we need to create both a following and a follower relationship. We just need to ZADD calls:

```
ZADD following:1000 5000
ZADD followers:5000 1000
```

Note the same pattern again and again. In theory with a relational database, the list of following and followers would be contained in a single table with fields like `following_id` and `follower_id`. You can extract the followers or following of every user using an SQL query. With a key-value DB things are a bit different since we need to set both the `1000 is following 5000` and `5000 is followed by 1000` relations. This is the price to pay, but on the other hand accessing the data is simpler and extremely fast. Having these things as separate sets allows us to do interesting stuff. For example, using ZINTERSTORE we can have the intersection of 'following' of two different users, so we may add a feature to our **Twitter clone** so that it is able to tell you very quickly when you visit somebody else's profile, "you and Alice have 34 followers in common", and things like that.

You can find the code that sets or removes a following / follower relation in the `follow.php` file.

Making it horizontally scalable

Gentle reader, if you read till this point you are already a hero. Thank you. Before talking about scaling horizontally it is worth checking performance on a single server. Retwis is *extremely fast*, without any kind of cache. On a very slow and loaded server, an Apache benchmark with 100 parallel clients issuing 100000 requests measured the average pageview to take 5 milliseconds. This means you can serve millions of users every day with just a single Linux box, and this one was monkey ass slow... Imagine the results with more recent hardware.

However you can't go with a single server forever, how do you scale a key-value store?

Retwis does not perform any multi-keys operation, so making it scalable is **simple**: you may use client-side sharding, or something like a sharding proxy like Twemproxy, or the upcoming **Redis** Cluster.

To know more about those topics please read our documentation about sharding. However, the point here to stress is that in a key-value store, if you design with care, the data set is split among **many independent small keys**. To distribute those keys to multiple nodes is more straightforward and predictable compared to using a semantically more complex database system.
