Posting or Deleting a Status Update
By Josiah L. Carlson, author of Redis in Action

In this article, based on chapter 8 of Redis in Action, author Josiah Carlson talks about Twitter user and status objects, which are the basis of almost all of the information in our application, and then deleting posts, which involves manipulating followers/following lists.

One of the most fundamental operations on a service like Twitter is posting status messages. People post to share their ideas, and people read because they’re interested in what’s going on with others. We’ll first show you how to create a status message as a prerequisite for knowing the types of data that we’ll be storing, and we will show you how to get that status message into a profile timeline or the home timeline of the user's followers.

**Status messages**

While user profiles store information about an individual, the ideas that people are trying to express are stored in status messages. As was the case with user information, we’ll store status message information inside a HASH.

In addition to the message itself, we’ll store when the status message was posted, the user ID and login of the user who posted it (so that if we have a status object, we don’t need to fetch the user object of the poster to discover their login name), and any additional information that should be stored about the status message. Figure 1 shows an example status message.

Figure 1 Example status message stored in a HASH
And that’s everything necessary for a basic status message. The code to create such a status message can be seen in the next listing.

**Listing 1 How to create a status message HASH**

```python
def create_status(conn, uid, message, **data):
    pipeline = conn.pipeline(True)
    pipeline.hget('user:%s@login', 'login') #1
    pipeline.incr('status:id:') #2
    login, id = pipeline.execute()

    if not login: #3
        return None

    data.update({
        'message': message, #4
        'posted': time.time(),
        'id': id,
        'uid': uid,
        'login': login,
    })
    pipeline.hmset('status:%s'%id, data)
    pipeline.hincrby('user:%s'%uid, 'posts') #5
    pipeline.execute()
    return id #6

#1 Get the user’s login name from their user ID.
#2 Create a new ID for the status message.
#3 Verify that we have a proper user account before posting.
#4 Prepare and set the data for the status message.
#5 Record the fact that a status message has been posted.
#6 Return the ID of the newly created status message.
```

There isn’t anything surprising going on in the status creation function. The function fetches the login name of the user, gets a new ID for the status message, and then combines everything together and stores it as a HASH.

We’ll now talk about making the status message visible to followers.

**Updating the status**

In this section, we’ll discuss what happens to a status message when it’s posted so it can find its way into the home timelines of that user’s followers. We’ll also talk about how to delete a status message.

You already know how to create the status message itself, but we now need to get the status message ID into the home timeline of all of our followers. How we should perform this operation will depend on the number of followers that the posting user happens to have. If the user has a relatively small number of followers (say, up to 1,000 or so), we can update their home timelines immediately. But for users with larger number of followers (like 1 million, or even the 25 million that some users have on Twitter), attempting to perform those insertions directly will take longer than is reasonable for a user to wait.

To allow for our call to return quickly, we’ll do two things. First, we’ll add the status ID to the home timelines of the first 1,000 followers as part of the call that posts the status message. Based on statistics from a site like Twitter, that should handle at least 99.9% of all users who post (Twitter-wide analytics suggest that there are roughly 100,000 to 250,000 users with more than 1,000 followers, which amounts to roughly 0.1% of the active user base). This means that only the top 0.1% of users will need another step.

Second, for those users with more than 1,000 followers, we’ll start a deferred task. The next listing shows the code for pushing status updates to followers.

**Listing 2 Update a user’s profile timeline**

```python
def post_status(conn, uid, message, **data):
    id = create_status(conn, uid, message, **data) #1
    if not id: #2
        return None

    posted = conn.hget('status:%s'%id, 'posted') #3
```

For Source Code, Sample Chapters, the Author Forum and other resources, go to [http://www.manning.com/carlson/](http://www.manning.com/carlson/)
Notice that we broke our status updating into two parts. The first part calls the create_status() function from listing 1 to actually create the status message, and then adds it to the poster's profile timeline. The second part actually adds the status message to the timelines of the user's followers, which can be seen next.

Listing 3 Update a user's followers' home timelines

```
POSTS_PER_PASS = 1000  #1

def syndicate_status(conn, uid, post, start=0):
    followers = conn.zrangebyscore('followers:%s'%uid, start, 'inf',  #2
                                  start=0, num=POSTS_PER_PASS, withscores=True)
    pipeline = conn.pipeline(False)
    for follower, start in followers:  #3
        pipeline.zadd('home:%s'%follower, **post)  #4
    pipeline.ziremrangebyrank('home:%s'%follower, 0, -HOME_TIMELINE_SIZE-1)
    pipeline.execute()

    if len(followers) >= POSTS_PER_PASS:  #5
        execute_later(conn, 'default', 'syndicate_status',  
                      [conn, uid, post, start])

#1 Only send to 1000 users per pass.
#2 Fetch the next group of 1000 followers, starting at the last person to be updated last time.
#3 Iterating through the followers results will update the "start" variable, which we can later pass on to subsequent syndicate_status() calls.
#4 Add the status to the home timelines of all of the fetched followers, and trim the home timelines so they don't get too big.
#5 If at least 1000 followers had received an update, execute the remaining updates in a task.
```

This second function is what actually handles pushing status messages to the first 1,000 followers' home timelines, and starts a delayed task using an API for followers past the first 1,000. With those new functions, we've now completed the tools necessary to actually post a status update and send it to all of a user’s followers.

Exercise: Updating lists

In the last section, I suggested an exercise to build named lists of users. Can you extend the syndicate_message() function to also support updating the list timelines from before?

Let's imagine that we posted a status message that we weren't proud of; what would we need to do to delete it?

It turns out that deleting a status message is pretty easy. Before returning the fetched status messages from a user's home or profile timeline in get_messages(), we're already filtering "empty" status messages with the Python filter() function. So to delete a status message, we only need to delete the status message HASH and update the number of status messages posted for the user. The function that deletes a status message is shown in the following listing.

Listing 4 A function to delete a previously posted status message

```
def delete_status(conn, uid, status_id):
    key = 'status:%s'%status_id
    lock = acquire_lock_with_timeout(conn, key, 1)  #1
    ```
While deleting the status message and updating the status count, we also went ahead and removed the message from the user’s home timeline and profile timeline. Though this isn’t technically necessary, it does allow us to keep both of those timelines a little cleaner without much effort.

### Exercise: Cleaning out deleted IDs

As status messages are deleted, “zombie” status message IDs will still be in the home timelines of all followers. Can you clean out these status IDs? Hint: Think about how we sent the messages out in the first place. Bonus points: also handle lists.

Being able to post or delete status messages more or less completes the primary functionality of a Twitter-like social network from a typical user’s perspective. But to complete the experience, you may want to consider adding a few other features:

- Private users, along with the ability to request to follow someone
- Favorites (keeping in mind the privacy of a tweet)
- Direct messaging between users
- Replying to messages resulting in conversation flow
- Reposting/retweeting of messages
- The ability to @mention users or #tag ideas
- Keeping a record of who @mentions someone
- Spam and abuse reporting and controls

These additional features would help to round out the functionality of a site like Twitter, but may not be necessary in every situation. Expanding beyond those features that Twitter provides, some social networks have chosen to offer additional functionality that you may want to consider:

- Liking/+1 voting status messages
- Moving status messages around the timeline depending on “importance”
- Direct messaging between a prespecified group of people
- Groups where users can post to and/or follow a group timeline (public groups, private groups, or even announcement-style groups)
Summary
In this article, we showed you how to post or delete a status update on Twitter. If you dig into chapter 8 of *Redis in Action*, you'll see that we've built the majority of functionality that makes a site like Twitter work. Though these structures won't scale to the extent that Twitter does, the methods used can be used to build a small social network easily. With a front end for users to interact with, you can start your own social network with your friends!

If there's one thing that you should take away from this chapter, it's that even immensely popular websites have functionality that can be built with the tools available inside of Redis.
Here are some other Manning titles you might be interested in:

- **Big Data**
  Nathan Marz and James Warren

- **Hadoop in Practice**
  Alex Holmes

- **HBase in Action**
  Nick Dimiduk and Amandeep Khurana

Last updated: June 16, 2013