FBFS: The Facebook Filesystem

Andy Russell

May 14, 2014

Abstract

Facebook allows users to search an incredible amount of information, including status updates, photos and "likes". However, this information is only accessible by clicking through multiple Web pages, and searching for a specific piece of information is sometimes difficult through Facebook's Web search.

In this paper, I describe FBFS, the Facebook Filesystem, a filesystem that provides access to a Facebook user's information. The implementation presents Facebook information such as the status updates and photos of friends as actual files on the user's filesystem. An added benefit of the filesystem is that many UNIX tools such as `grep` and `find` are available to search for Facebook content using regular expressions or other advanced queries, something not possible with Facebook's own Web search. FBFS leverages Facebook's API (the Graph API) to provide information when needed, without storing Facebook data on a user's computer. I have taken care to provide a useful interface that takes advantage of many filesystem features to provide a positive user experience.

1 Introduction

Facebook is a social networking website with more than 1 billion active monthly users [**?**]. Facebook's popularity brings a diverse set of users with it. These unique users interact with the website in many different ways: through the Web interface on their computers, via apps on their tablets or smartphones, or with text messages on traditional cell phones. For those who use Facebook on their computer, there are also various ways of searching the vast amount of information available: a user might click through links in the web interface until their query is resolved, or utilize the search bar at the top of the page.

The goal of FBFS is to provide an alternative interface to Facebook by simulating a filesystem on a user's computer. This allows the user to access the information that is available to him or her on Facebook while taking advantage of the familiar hierarchical interface of a filesystem.

2 Related Work

FBFS is not the first project of its kind. I found two similar projects in my initial research.

The first project, "Facebook Filesystem", is mainly tongue-in-cheek [**?**]. The filesystem is implemented in Perl. The program allows users to access their Facebook photos through a filesystem interface. Additionally, the system allows users to add comments to the photos, which are then embedded in the photo's binary data. However, this creates a problem when the photo is uploaded back to Facebook. Because Facebook compresses photos on upload, the binary data appended to the end of the photo corrupts the rest of the image when displayed on the site. Though the filesystem is intended as a joke, it raises an interesting question about the design of a Facebook Filesystem: the question of how to deal with metadata. I did not install this filesystem to my computer, as the author himself warns that the filesystem is likely to corrupt your Facebook photos.

The second project I examined was also called "FBFS" [**?**]. This filesystem is implemented in C. It has similar goals to my implementation of

FBFS: the earlier program aims to provide an interface that allows UNIX utilities to interact with information on Facebook in a meaningful way. However, the implementation differs from mine in a number of ways. It requires a user to run a Ruby Web server to create an access token, which the user then must store in a predetermined directory. I disagree with this method of storing the access token (explained further in Section 4). In addition, the actual file system layout of the C implementation mirrors the Facebook API more closely, which can make it more difficult for the user to interact with the filesystem, because they have to remember the structure of the Graph API in order to interact with the system. Though I attempted to install FBFS to my computer, I could not get the binary to link properly.

Both implementations provided useful insight into some of the design decisions that I had to make for FBFS.

3 Background

3.1 The Graph API

Facebook provides a powerful API for third-party applications to interact with their website, hereafter referred to as the Graph API [**?**]. The Graph API is structured in terms of "nodes" and "edges" though they do not mean quite the same thing as the graph terminology they are inspired by. The API is HTTP-based and responds with JSON, allowing developers to use existing tools to interact with the API in a language-independent way. The syntax for requests is summarized in Figure 1.

A node represents an object on Facebook, and an edge represents some kind of information about that node. Additional parameters for the request may be added on as key-value pairs at the end of the URL. If the request requires special permissions that were requested when the application received the access token (discussed in Section 3.2), the access token must be appended at the end of the request as a URL parameter.

Upon receiving and processing a request, the API will respond with a JSON object that contains a single field "data", that maps to object representing the information requested. If the request is malformed or the application that spawned the request has insufficient permissions to access the data requested, the response will instead have a single field "error", that maps to an object containing an error code and a brief message. I have included a few examples of API requests and responses in Table 1 to help illustrate typical API interaction. Request headers, the hostname, the HTTP version, and the access token parameter are all omitted from the request for simplicity, though each is required when actually interacting with the API.

I have summarized some further examples of API requests and the effect they have on Facebook in Table 2. Though the API is quite expressive, it is possible to obtain further granularity in the request by using FQL, a SQL-like query language that allows complicated queries against various "tables" such as friends and photos.

3.2 Facebook Applications

For a program to communicate with Facebook, it must be associated with a Facebook application. A programmer creates an application by logging into Facebook's developer portal and choosing a name. From here, the developer can access the "app ID", which is required to identify a client at login, and the "app secret", which is used for requesting special application-level permissions from a client.

Once the Facebook application has been created, the client login process is quite simple. First, the client directs a browser to a URL containing the client ID and any additional permissions that the application needs to function. The user is prompted to enter their username and password. If the login is successful, then the user reviews the permissions requested by the program and either accepts or declines the request to install the application. The browser is automatically redirected to a URL that indicates whether the login was successful or not. If successful, the URL will contain an access token that is required for every login.

Figure 1: Graph API Request Syntax

`GET https://graph.facebook.com/<node>/<edge> HTTP/1.1`

Facebook deals with permissions in a very flexible way. An application may request additional permissions during normal execution by following the same protocol as the login process. A user may choose to accept or deny these additional permissions, and Facebook suggests that applications only request permissions that are absolutely necessary. In addition, there are permissions that a user may choose to accept or deny for all applications, such as allowing access to their statuses from friends' applications.

4 Design

FBFS is designed to be as user-friendly as possible. This is achieved in part by adhering to the UNIX philosophy. To satisfy both constraints, there are a number of design trade-offs that were made with the goal of making the filesystem easy to use.

One of the problems encountered by this filesystem is the incredible amount of metadata that each object on Facebook has associated with it. For example, a photo album contains photos, each of which has comments, which have likes, etc. In the initial phases of design, I considered storing this metadata in a structured file containing human-readable markup such as YAML [**?**]. However, such a structure would probably make it harder to search the files or use them as input to programs. This also would violate the tenet of the UNIX philosophy which states that data should be stored in flat text files [**?**]. Therefore, FBFS compromises on this point and chooses to store Facebook information such as the text of a status inside a flat file, while keeping comments in a separate, structured file.

Another design tradeoff that was necessary was a deviation from the Graph API structure. Though the Graph API provides a hierarchical structure similar to a filesystem in many cases, there are a number of ways in which it differs. For example, consider that any object on Facebook may be used as a node. It would be both unwieldy for a user and impractical for the developer to fill the root directory with every node that could be available. Therefore, I have decided to use the various edges of the graph (statuses, photos, friends, etc.) as folders stored in the root directory. This makes it easy for a user to see all of their statuses by listing the status directory. The mapping of various API requests to filesystem operations is summarized in Table 3.

FBFS requires a user to login every time they mount the filesystem. This ensures that the access token is fresh and valid, but it also burdens the user with entering their username and password more often than is convenient. Though it would be possible to store the access token between sessions and automatically log the user in, this would create a security risk. A program could hijack the FBFS access token (especially if it were stored in a well-known location), and send malicious requests.

Since FBFS brings Facebook from the Web browser to the filesystem, there is a completely different set of features available. Where the website uses a page covered with links, the filesystem uses a directory with multiple directory entries. One of the design goals of FBFS is to use filesystem features in a user-friendly and predictable way. An example of such a feature is the UNIX permissions system. This attribute can be mapped to Facebook's own permissions system. As discussed in Section 3.2, the Facebook permissions system is very flexible. FBFS aims to respect the permissions of the user and requests minimal permissions at login. For example, if a user denies the Facebook application access to their photos, the `photos` endpoint in the root directory will have neither read nor write access for the user. If they later allow such permissions, the permissions bits will be updated accordingly.

The filesystem also utilizes symbolic links to link mutual friends in a friend's directory. Imagine a user who has two friends, A and B. The user is friends with both A and B, and A is also friends with B. The user might type `ls /friends/A/friends/` and see B's folder in the directory listing. The user could conceivably recurse through the directory structure infinitely by following the path `/friends/A/friends/B/friends/A...` forever. By reporting `/friends/A/friends/B/` as a symbolic link to `/friends/B/`, the problem is solved. If the user wishes to recurse infinitely, they are allowed to do so, but the filesystem treats the paths as no more than two levels deep.

These features also work in tandem. FBFS sets the permission bits of the directories of non-mutual friends-of-friends to zero. Then, when a user lists a friend's `friends` directory, it is easy to discern which friends are mutual, because those friends will be symlinks, while other, non-mutual friends will be normal directories.

One of the more difficult decisions to make was the naming of files on the filesystem, in particular for the statuses. Originally I used ISO-8601 [**?**] timestamps for the names. However, this proved undesirable because the names were hard to read and the timestamp information was already stored in the file metadata. Most other options such as using the first n characters of the status also did not work because they were not necessarily unique between multiple statuses. The ultimate decision was to use the node ID of the status for the filename. Though the node ID is not easy to read, it both guarantees uniqueness and simplifies metadata lookup, as described in Section 5.

The result of such tradeoffs is that the filesystem is usable and fast in the average use case.

5 Implementation

FBFS is implemented in two parts: the filesystem, and the Facebook application. The Facebook application, as discussed in Section 3.2, is required for interacting with the Facebook API but requires little developer customization. The filesystem is where most of the implementation effort was focused. The filesystem is written in C++, using a variety of libraries.

Since Facebook does not provide an official C++ SDK, I had to resort to manually building a login flow [**?**] (described in Section 3.2). When the user mounts FBFS, the program opens a Web browser for the user to interact with Facebook. Initially, the program opened the user's default browser (through `xdg-open`). Though convenient for the user, this method was impractical because it was impossible to inspect the current URL and obtain the access token from an external program. Instead I opted to open a browser using QtWebKit, a class provided by the Qt Application Framework [**?**]. Opening the browser in this way also allows the program to prevent the user from entering custom URLs into the browser, which simplifies checking whether the returned access token is legitimate. If the access token is expired or the user refuses to install the Facebook application, the program terminates.

Once the application is authenticated, it needs to send a large number of requests to the Graph API. These requests are sent with the curlcpp library [**?**], a C++ wrapper around libcurl [**?**]. Responses are parsed by the JSON Spirit library [**?**]. JSON Spirit provides a sensible mapping between JSON types and C++ STL types. For example, JSON objects have an interface similar to `std::map`, JSON arrays map to `std::vector`, and so on.

The biggest performance impact to the filesystem is the number of Graph API calls that the client must make. This problem is mitigated in two ways. First, the number of required API calls have been reduced by using node IDs as the file names. By knowing the node ID, it is possible to look up all of the metadata about a Graph object in a single API call. Secondly, subsequent API calls have been sped up by implementing a caching mechanism. Every call to the Facebook API is stored in a map that associates triples of the form $(node, edge, parameters)$ to the JSON returned by the API. The cache is invalidated periodically, and there is also a method to skip the cache on specific API calls if fresh data is needed. This drastically cuts down on the num-

ber of API calls that must be made. This is particularly important because FUSE calls the `getattr` function quite often, and it is necessary to fetch information from the API to determine file metadata such as size or modified time.

The complete code for FBFS is hosted on GitHub [**?**].

6 Evaluation and Future Work

FBFS has generally reached its initial goals. The filesystem is usable and searchable with classic UNIX tools.

Performance is the biggest problem currently facing FBFS. According to some testing with Facebook's Graph API Explorer [**?**], an average request takes about 250ms, with long requests taking over a second. This creates an enormous performance impact for even the most basic filesystem operations. Consider listing a user's status directory with a cold cache. The initial `ls` invocation will call FUSE's `getattr` and `readdir`. Then, the filesystem will have to determine the metadata for each entry in the directory. Assuming that the API returns the first 25 statuses, then a single directory listing can take over 10 seconds! Though subsequent calls to these status nodes would be cached, a potential optimization would be to cache based on the node ID instead of the request itself. This would make lookups far faster, and also more flexible, as a single edge call such as the one made by `readdir` could cache up to 25 nodes, making the subsequent `getattr` calls nearly instantaneous. I have included some performance measurements of FBFS with a cold and hot cache in Table 4. The measurements were made with the UNIX `time` command. Though caching is effective, notice the variability in the API response time, even for the same request.

Another user-friendly change that could be made in the future is to name the files with more easily-typable names than their node IDs. However, to keep the performance benefits from the naming scheme described in Section 4, it would be necessary to keep a map from the user-displayed name to the node ID for every object in the filesystem, which would increase the memory overhead.

Due to the sheer amount of information available on Facebook, it is a Herculean task to account for every type of object and thus create a complete Facebook filesystem. Though at the moment FBFS only supports viewing friends, statuses, photos, and albums, with more work it would be possible to support more endpoints. Each endpoint requires node-specific code in `getattr` and `readdir` in order for the information to be available.

In addition, since Facebook does not offer an official C++ SDK, I hope to release my Graph API wrapper as a standalone project for use in other C++ applications, as the existing unofficial SDKs are out of date.

7 Conclusion

Facebook is an exceedingly popular social network which provides an incredible amount of information to its users. FBFS allows users to interact with this information in a new way and enables users to utilize the power of a filesystem interface to consume and search this information. By bringing the power of the Graph API to the hands of the user, FBFS brings additional functionality to any Facebook client.

Table 1: Sample Graph API Requests

Request	Response
`GET /me/statuses`	`{"data": [ { "message": "Hello World!", "id": "123456789", ... other fields ... }, ... other status objects ... ] }`
`GET /123456789`	`{"data": { "message": "Hello World!", "id": "123456789", ... other fields ... } }`

Table 2: Graph API Examples

Request	Action
`GET /me`	Returns the current user's basic profile information.
`GET /me/friends`	Returns a list of objects containing IDs, names, etc. of the current user's friends.
`GET /`*`friend-id`*`/statuses?fields=message`	Returns an array of objects containing only the text of the last 25 statuses made by a friend with ID *friend-id*.
`POST /me/feed?message=`*`status`*	Posts a status on the current user's feed with the text *status*.
`DELETE /`*`photo-id`*	Deletes a photo with the id *photo-id* from the current user's profile.

Table 3: REST vs. Directory Hierarchy

Graph API	FBFS
`GET /`*`user-id`*`/statuses`	`ls -l /friends/Andy\ Russell/status`
`GET /`*`status-id`*	`cat /friends/Andy\ Russell/status/`*`status-id`*
`POST /me/feed?message=`*`status`*	`echo "`*`status`*`" > /status/post`
`DELETE /`*`photo-id`*	`rm /photos/`*`photo-id`*

Table 4: FBFS Performance

Operation	Cold Cache	Warm Cache
`ls friends`	1.072	0.079
	1.643	0.101
	0.862	0.102
	0.803	0.038
	0.833	0.100
`ls status`	10.882	0.225
	10.698	0.216
	4.642	0.235
	10.363	0.249
	22.313	0.346
`cat status/10151702452258380`	0.307	0.146
	0.137	5.849
	0.150	0.140
	0.362	0.350
	5.645	0.127