EXHIBIT 2

FacebookPlatformWhitePaperIX Last modified Tuesday, March 24, 2009 at 4:55pm by Dave Fetterma

IX. Development Implications

The Facebook Development Platform does not exist of its own merit; it relies on both user-facing Facebook concepts (e.g. "Facebook events") and the engineering necessary to support it (e.g. the code to support "events"). The public API is not a replacement for users, groups, events, photos, and the like; it is merely a broadening of the audience for those pieces of information. If the concept of an event were to change in the Facebook application, it would not do justice to Facebook, nor the developers who come to depend on both on their users' notion of a Facebook event and the technical specification of event data, to fail to update the notion of group for the outside community.

Therefore, the deployment of the public API implies that the data must retain a certain integrity otherwise not specifically required on the site. The development implications of the public API are then twofold:

- The integrity of the data must be maintained in a rigorous way. This means, there must be dedicated testing of the API when any code in the Facebook logic changes, to avoid passing bugs to everyone dependent upon the API.
- The data interface must be maintained and versioned tightly. Even the smallest change could disrupt the site of an outside partner, who we must (conservatively) assume is tightly coupled to every aspect of our functionality.

1. Testing

Let us say we are changing a piece of Facebook functionality for www.facebook.com, say search, during daily operations. This usually involves the search developers making a change, doing sandbox testing, and verifying to the best of their knowledge that the code change breaks nothing, and works as intended. Before pushing the code to live servers, the changes are copied to beta servers on inyour.facebook.com, and available Facebook engineers of all sorts are told to "bang around" on this feature to make sure there are no (obvious) bugs. The code is then pushed live. Invariably, bugs are found in that (or any other) feature, they are fixed, and the process iterates until the level of bugs is sufficiently small. This 'iterative' process serves as our main testing strategy. If bugs exist, they are visible on the Facebook site for a hopefully short time before they are fixed. In the normal case, faulty Facebook display and business logic affects the site. In the worst case, faulty Facebook logic affects the data, which is largely unrecoverable.

For all its faults, this strategy at most affects the Facebook site. The Development Platform primarily exists to enable outside applications to build off the Facebook site. A bug unchecked then could affect or ruin someone else's application, or all outside applications, which does much more damage to our relationships and reputation, not to mention the applications we empower through the API. For this reason, Facebook changes require a defined and dedicated test suite and process.

We are currently developing this suite, which needs to run at an acceptable level by the time of full release. However, we need to make sure that for all changes, this suite is run at the time of every push or hotfix affecting non-display logic not strictly in the www.facebook.com display layer.

The main drawback here is that our development and push process is slowed, and we could lose some agility. The resounding positive, however, comes when we have a dedicated testing process for the API, which in turn, verifies a good portion of the www.facebook.com functionality as well.

We need this test suite integrated into our daily development process, and visible to the entire engineering organization, beyond those directly responsible for its maintenance on the Platform engineering team.

2. Interface Versioning

Even if the functionality of www.facebook.com is not broken by a completely unintentional bug, alterations to the scheme of the data contained defining, say, a "Facebook group", can affect the public API if they are not properly communicated. The test suite comes in handy here, too, since these changes, at the time of push, will break the tight interface exposed by the API. However, we need a way to build on top of the existing contract we give out to applications developed on the API.

The first part is *communication*. Again, this is a process change. Changes to the data provided by our key public APIs need to be communicated to the Platform engineering team. We need a process to facilitate this.

The second part is *technical versioning*. If a data change meets with universal approval, the API either needs to back port its functionality to maintain something of an old interface, support multiple versions of the API contract, or be able to effectively drop old versions. This responsibility falls to the engineers of the public API, the Platform team.

First, simply adding to the API's functionality requires none of these measures. Even in a strict interface, adding members orthogonal to existing ones does not ruin the dependent code in any meaningful way. If the first edition of the API shipped without, say, notes are added and the API adds a way to interact with them, these new separate procedures can be added without fear.

Back porting is suitable when the change is very small. For instance, if the data structure for a user's network affiliations is changed from including a field for a network key to containing a substructure containing that key, the public API should translate that to maintain the old interface. This involves some work but does not engender inherent complexity.

Multiple versioning is the best choice when functionality has changed significantly for the better. The new interface has been deemed more useful, faster, cleaner, etc. and the API users should also enjoy that goodness. In order to not break applications using the now 'old interface', both old and new versions can be supported. This is a technical challenge, but entirely solvable within the Platform engineering organization. We are trying to achieve version support right from the first release.

Dropping old versions occurs after multiple versioning for a period of time. Partners are notified of the change, the date for dropping support for old versions, and that change happens. Naturally, we try to avoid this, especially for large partners with inflexible engineering, or client applications that have already been downloaded. Good planning is necessary to make sure this option is avoided, but there are times when it is necessary. Supporting dozens of different versions remains much worse than requiring our partners to evolve just as we do.

The Three-Version Queue

One elegant plan for combining these three features together is a three-version queue system. At any time, we maintain (at least) versions A, B, and C, where A, B, and C are some version numbers like 1.0.1, 1.0.2, 1.0.3. These correspond to ideas like "legacy", "current", and "latest". Version A additionally has an expiration time associated with it. When version A expires, the addition of version D immediately drops version A, and the timer begins on version B, the new "legacy" version. A reasonable timeout, especially once large partners with fixed release schedules enter the picture, would be on the order of 90 days.

Plaintiff's Trial Exhibit
PTX-145
Case No. 08-CV-00862

This *multiple versioning scheme* incorporates *backporting* for all but the latest version, and *drops old versions* on a public schedule. The developer application selects the interface version within the request protocol (likely an element in the XML procedure call).

Prev Section <<

Next Section >>

(touch)