Mark D. Hawker

The Developer's Guide to Social Programming

Building Social Context Using Facebook, Google Friend Connect, and the Twitter API

Developer's Library

Library of Congress Cataloging-in-Publication Data:

Hawker, Mark D.
The developer's guide to social programming : building social context using Facebook, Google friend connect, and the Twitter API / Mark D. Hawker.
p. cm.
ISBN 978-0-321-68077-8 (pbk. : alk. paper) 1. Online social networks. 2. Entertainment computing. 3. Internet programming. 4. Google. 5. Facebook (Electronic resource) 6. Twitter. I. Title.
HM742.H39 2010
006.7'54—dc22
2010020866

ISBN-13: 978-0-321-68077-8
ISBN-10: 0-321-68077-4

Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.

First printing, August 2010

Contents at a Glance

Table of Contents

IV: Putting It All Together

Preface

The World Wide Web is in constant flux and, since the introduction of utilities such as Facebook and Twitter, has only recently had social interaction at its core. Currently, Facebook and Twitter have more than 400 million active users, and the Facebook Platform alone is integrated with more than 250,000 websites and applications, engaging over 100 million Facebook users each month. These numbers continue to increase each day. Another dominant force is Google, who introduced their Friend Connect, which enables users to add social functionality to any of their websites. All three companies continue to roll out massive changes to their development platform, rendering previous best practices obsolete.

However, just knowing the technical aspects of each platform is not a guarantee that it will succeed. It is important to also see how each is distinct and to prepare you for changes through examples and sample code. The purpose of these examples is to provide a springboard to build applications on, so there is plenty of room for extending and adapting to suit your own needs. This book is one of the first of its kind to bring together three of the most popular social programming platforms under one hood. Welcome to social programming.

Who This Book Is For

This book is written for beginner or intermediate developers who are comfortable with PHP and the major technologies of the Web: (X)HTML, JavaScript, and Cascading Style Sheets (CSS), as well as Atom, JavaScript Object Notation (JSON), Really Simple Syndication (RSS), and Extensible Markup Language (XML). The reader should also have access to a web server, such as Apache or Internet Information Services (IIS), to test code examples.

No prior experience of social programming is required, although some familiarity and active user accounts with Facebook, Google, and Twitter is assumed. To be a good developer for a platform, it helps to understand it from a user's perspective.

This book will help the reader understand what makes a good Facebook, Google Friend Connect, and Twitter application; explain and show how to use the core technologies of each platform; and build your confidence to develop engaging social applications.

How This Book Is Structured

This book is divided into four main parts:

Part I, "Twitter," provides an overview of the methods, authentication workflows, and components of the Twitter API. It explains what is contained within the Twitter API, including search, retweets, lists, and geolocation using code examples supported by a PHP client library, twitter-async.

Part II, "Facebook Platform," provides an overview of the service, including authentication, sharing, commenting, and publishing. A sample application is created, Test Tube,

highlighting key features of the platform through both client- and server-side scripting using the Facebook Platform.

Part III, "Google Friend Connect," showcases the service and its integration with OpenSocial through client- and server-side scripting and the creation of a Google gadget. A sample application, Color Picker, is created to demonstrate Google Friend Connect in action.

Part IV, "Putting It All Together," pulls each of the three social platforms together into a coherent whole and demonstrates how to create your very own microblog from scratch. A sample application, Sprog, is created using a popular web application framework, CodeIgniter, which is extended using select functionalities from Twitter, Facebook, and Google Friend Connect.

Contacting the Author

If you have any questions or comments about this book, please send an e-mail to socialprogramming@gmail.com. You can also visit the book's website, http://www.socialprogramming.info, for updates, downloadable code examples, and platform news. An active code repository will be maintained, http://github.com/markhawker/Social-Programming, which you can use to post issues you have with the code and to download future updates.

8

Application Discovery, Tabbed Navigation, and the Facebook JavaScript Library

Facebook can be used as a mechanism for sharing content, commenting, and stream publishing, as you learned in Chapter 7, "Using Facebook Connect for Sharing, Commenting, and Stream Publishing." However, the Facebook environment contains three other ways in which users and their friends can interact: application dashboards, which focus on the discovery and reengagement of games and applications; counters, for alerting users that they need to take action on an application or game (perhaps taking their next turn) or that a report is ready for them to view; and application tabs, which can be shown on a user's profile alongside other profile information. These three channels can be used by a Facebook Platform application to engage users both within and outside of Facebook.

This chapter explores how you can use dashboards in your Facebook Platform application through the Dashboard API. Through the Dashboard API, you can post news items to a user's dashboard, promote friends' activities, and utilize activity counters. The second part of this chapter focuses on application tabs as a way of sharing your application's information with users and their friends. Following the deprecation of profile boxes, application tabs are the only mechanism for enabling users to personalize their profiles and showcase their favorite applications. This section includes details about how to configure, install, and develop an application tab through the use of "Mock AJAX". The final section showcases Facebook JavaScript (FBJS) and how you can use it for events, animations, and Facebook dialogs.

Application Dashboards and Counters

> **Dashboard API**
>
> At the time of this writing, the methods from the Dashboard API were not available to test and could be subject to change. When the Dashboard API becomes fully available, examples will be added to this book's code repository. A blog post will also be added to the book's website denoting that the functionality in this section is available.

Because of the popularity of social gaming applications on Facebook, their recent redesigns have started to put more emphasis on highlighting specific features for games. There are now two types of "dashboards": Games (http://www.facebook.com/?sk=games) and Applications (http://www.facebook.com/?sk=apps). These are accessible via a user's home page alongside bookmarks. The goal of each of the dashboards is to make it easier for Facebook users to access games or applications that they or their friends have recently used and to discover new applications through their friends or the Application Directory (see Figure 8.1).

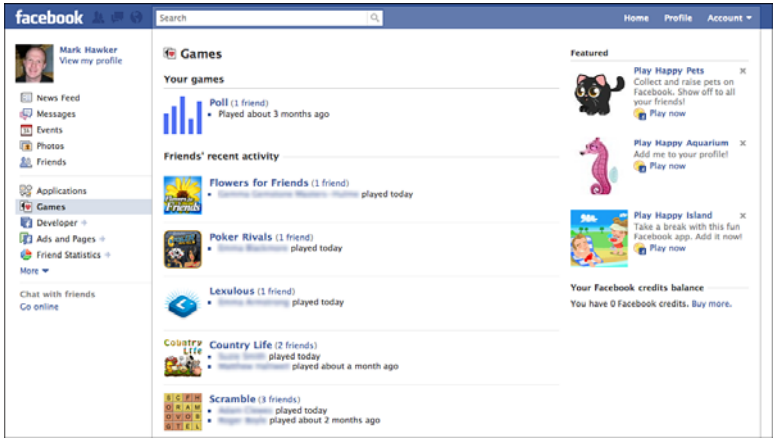

Figure 8.1 Screenshot of the Games dashboard.

Various key features are available within the dashboards:

- Recently used applications or games display right at the top so that users can quickly and easily find applications they use on a daily basis. The number of friends who also use the application is also highlighted next to each application's title.
- News items can be used to allow applications to communicate with users either to display news to all users or alert individual users that they need to take action. For example, a game news item may say "It's your turn to play, Mark!" You can also use news items to mention a user's friends and invite them to play a game with you.

- A user's friends' recent activity is shown, which is used to promote applications and games that a user might not have installed. This can also be toggled to display the activities that an individual has recently completed, which can be privacy controlled.
- A list is maintained of all friends who recently interact with applications that appear below the activities. These take the form of a list and are updated dynamically based on usage.
- The legacy Facebook Application Directory is displayed right at the bottom of the dashboards for searching for applications in particular categories. When submitting your own application to the directory, this will be the category or categories that you have provided.
- Facebook also runs features on particular applications or sponsored applications, which are shown to the right side of a profile. These are generated by combining a users' and their friends' activities to suggest the most suitable applications or games to their profile.
- Counters can be shown alongside an application's name for games or applications that a user has bookmarked. These are discussed further in the "Games and Applications Counters" section, later in this chapter.

When submitting an application to the directory, a developer will choose whether the application should be listed as a "game" or as a regular "application." This designation dictates which dashboard it will be placed within. Both dashboards contain the same functionality and so differ only in content. A new Dashboard API was released in February 2010 to encompass all the features of dashboards (the subject of the remainder of this section), including adding to news and activity streams and updating counters.

News and Activity Streams

As you have seen in the Games and Application dashboards, Facebook has concentrated a lot of their efforts on keeping users updated as to what they and their friends are up to. One of the main ways in which this is achieved is through activity streams. Activities are reported on the Games and Application dashboards in two distinct ways, through news and activities:

- News items can be set to display global and personal items to an individual or set of individuals. These could be that a new feature has been added to your game or application or if a friend has initiated an action involving a particular user.
- Activity items display actions performed specifically by the individual that appear in that individual's stream but could also reference one of their friends. In which case, that individual's activity will also appear in a friend's news items.

The methods for each of these streams are similar to those regarding stream publishing discussed in Chapter 7. The only real difference here is that news and activities are restricted to the dashboards rather than the user's stream, which helps to reduce unnecessary clutter.

Working with News Items

News items are a way of sharing announcements with your users or for indicating that a friend has performed an activity that has referenced them. There are two types of news items, global and personal, depending on what method was called to create the item. Facebook displays just two news items within either the Games or Applications dashboard, and so they also provide a convenient method to clear news items from a user's stream.

Adding News Items

> **Dashboard API Naming Conventions**
>
> Although Facebook lists these methods as including `add` in their name, this might change to `set` in the future. In the most recent version of the Facebook API PHP client library, the `dashboard.addNews` method was actually `dashboard.setNews` but returned an error when executed.

News items can be added using the following methods either individually, globally, or for multiple individuals using the following:

- `dashboard.addNews`
- `dashboard.addGlobalNews`
- `dashboard.multiAddNews`

Each method requires a slightly different set of parameters, such as providing a `uid` (which is that of the user whose dashboard you are updating) for individual news and which is not required for global news items. For updating multiple users, an array of `uids` is required instead. This array contains a number of user identifiers that require updating. Note that you cannot set multiple messages for each of these individuals, so each news item will be the same for each of the identifiers you provide. An array of up to eight `news` items is also required. This must contain a `message` and an optional `action_link` that includes `text` and a `href`. If you want, you can also supply an optional `image` parameter. This must be an absolute URL that is formatted as a 64x64px square. An example of each method is shown here:

```
$user = $facebook->get_loggedin_user();
$users = array("1", "2", "3");
$news = array(
 array(
  "message" => "Hey, {*actor*}. Your friend @ just invited
  you to play chess.",
  "action_link" => array (
   "text" => "Play Now!",
   "href" => "http://myfacebookapp.com/?game=chess"
  )
 )
);
$global_news = array(
```

```
 array(
  "message" => "Hey, {*actor*}. There is a new game to play, chess.",
  "action_link" => array (
   "text" => "Play Chess!",
   "href" => "http://myfacebookapp.com/?game=chess"
  )
 )
);
$image = "http://29.media.tumblr.com/avatar_abad48dbd089_96.png";
$individual_news = $facebook->api_client->dashboard_addNews($user, $news,
$image);
$global_news = $facebook->api_client-
>dashboard_addGlobalNews($global_news, $image);
$multi_news = $facebook->api_client->dashboard_multiAddNews($users, $news,
$image);
```

If successful, the `$global_news` and `$individual_news` items will return a `news_id` if the call succeeds, and the `$multi_news` item will return an associative array of `uid` keys that contain either a `news_id` if successful or `false` if unsuccessful. These `news_id` values are important and should be stored because they will be required if news items need to be cleared from a dashboard. In addition, two conventions were demonstrated in the message values: You can use the `{*actor*}` token, which is also available within stream attachments, to be rendered as the user whose dashboard is being updated; and you can use `<<USER_ID>>`, where `<<USER_ID>>` can be replaced by any user identifier. In your own applications, this would form part of a two-stage process of updating an individual's activity stream but also updating the news streams of that user's friends that he or she was playing against or wanting to update.

Clearing News Items

As with adding news items, three methods enable you to clear updates that have already been created by an application. Clearing individual news will not remove global news and vice versa, and so these methods may be used alongside each other:

- `dashboard.clearNews`
- `dashboard.clearGlobalNews`
- `dashboard.multiClearNews`

All of an individual's news items can be removed by using the `dashboard.clearNews` method and supplying their `uid` as the single required parameter or by additionally passing in an array of `news_id` values. For global news, the `dashboard.clearGlobalNews` method can be called without any parameters to remove all news or can include an array of `news_id` values similar to the individual news item method. Clearing multiple individuals' news items is slightly more complex. Here is an example assuming that the `$multi_news` parameter that was presented in the "Adding News Items" section above returned the following:

```
$multi_news = array(
 "1" => 111,
 "2" => 222,
 "3" => 333
);
$ids = array(
 "1" => array("111"),
 "2" => array("222"),
 "3" => array()
);
$removed_multi_news = $facebook->api_client->dashboard_multiClearNews($ids);
```

A successful response from the individual and global methods is an associative array of `news_id` keys and Boolean values depending on whether the news item has been removed. When you are removing multiple individuals' news items, an associative array will be returned equivalent to the individual and global methods if `news_id` values were supplied. Otherwise, if no `news_id` values were supplied (such as the last `$ids` parameter), an associative array will be returned containing the `uid` as the key and a Boolean value of whether the news item was removed or not.

Getting News Items

The final sets of methods are used to extract a user's or group of users' news streams. Simply put, these methods provide you with the original `news` and `image` values that were set when adding news items. The method names are as follows:

- `dashboard.getNews`
- `dashboard.getGlobalNews`
- `dashboard.multiGetNews`

These methods prove particularly useful should you not want to store `news_id` values within your database of file stores.

Working with Activity Items

Unlike news items, activity items are an experimental feature and may be removed by Facebook in the future. Activity streams are used to broadcast to a user's friends what that user been up to within a game or application (for example, posting high scores or whether the user has uploaded new files or photos). There are only three methods for working with activity items, and these cannot be called for multiple individuals like news items:

- **`dashboard.getActivity`**

  This method will return the latest 100 activities recorded for the current user. The method can be called with an optional `activity_ids` array if you have recorded each `activity_id` for your users.

- **dashboard.publishActivity**

  This method works in exactly the same way as `dashboard.addNews`, but rather than being a `news` object, it is an `activity`. The same conventions for using `{*actor*}` and `<<USER_ID>>` tokens can be used when setting activity items. Successful publishing of an activity will return a numeric `activity_id`.

- **dashboard.removeActivity**

  Activities can be removed by supplying an array of `activity_id` values, which will return an associative array of `activity_id` keys and a Boolean value indicating success or failure.

When setting up your application in Chapter 5, "An Overview of Facebook Platform Website Integration," you may have noticed a setting called Hide User Activity within the "Advanced" tab. This setting can be checked if you think that your application will generate activities that a user might want to keep private and not share with friends. Although further details were not available at the time of this writing, Facebook intends to give users sufficient control over which news and activity items they both send and receive. Like items being posted to their stream, it may be that they want to inform certain friends of their activities but exclude others.

Games and Applications Counters

Before an application or game can utilize counters, it must first be bookmarked by the user. This can be done from within Facebook using the links provided on each of the dashboards. However, it can also be facilitated through embeddable `<fb:bookmark>` FBML and XFBML tags. A bookmark URL must be set. You can find this within the "Basic" tab of an application; otherwise, the application's connect URL or canvas page URL will be used. For Facebook Platform applications, you can set the `type` attribute of the button to `off-facebook`, which will render a blue button in place of the standard gray used within canvas applications. Upon clicking the button, users are prompted with a dialog box to add the application to their profile (see Figure 8.2).

If a user has already bookmarked your application, the button will not appear. You can also check this by querying the `permissions` FQL table, as follows:

```
$bookmarked = $facebook->api_client->fql_query(‘
 SELECT uid, bookmarked
 FROM permissions
 WHERE uid = "‘.$official_user.’"
‘);
```

The result of this FQL query will be either a `1` or a `0` that can be extracted by using `$bookmarked[0]["bookmarked"]`. New bookmarks will appear underneath the links to the Games and Applications dashboards and can be rearranged by users after clicking the "More" link below their bookmarks. After an application has been bookmarked, you can start exploiting the features of counters via the Dashboard API.

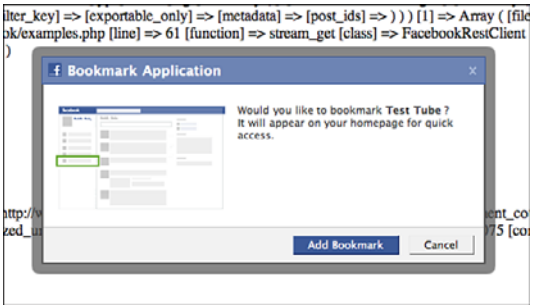

Figure 8.2 Example bookmark dialog for the Test Tube application.

There are two types of counter methods. One type of method enables you to update an individual's counter. The other type of method can be used to update a number of individuals' counters. Users could utilize this to let a group of friends know of an action they've taken in a game and that it is now their turn. There are four methods for updating the first type of counter for individuals:

- `dashboard.decrementCount`
- `dashboard.getCount`
- `dashboard.incrementCount`
- `dashboard.setCount`

These methods can be run either using the logged-in user's credentials or by supplying a `uid` alongside your application secret. Unlike internal Facebook applications or games, when using website integration you must ensure that every time a user visits your bookmark URL that the user's counter is reset to zero. For applications that want to update a group of individuals' counters at the same time, the second type of counter method, a number of batch methods are available:

- `dashboard.multiDecrementCount`
- `dashboard.multiGetCount`
- `dashboard.multiIncrementCount`
- `dashboard.multiSetCount`

These batch methods all request that an array of `uids` be supplied and will return an array of `uids` as the key and a Boolean value for whether the request was successful. It is suggested that when users visit your application, either on a canvas page or via an external website, that their counter is set to zero to ensure that users do not get confused as to what actions they are required to take.

Navigating and Showcasing Your Application Using Tabs

In the early days of Facebook, a number of "integration points" were available to developers to showcase their applications. These integration points included profile boxes, news feeds, and notifications. As a greater mass of developers started using the platform, Facebook quickly became a dumping ground for spam because insufficient controls and policies failed to prevent malicious developers abusing the platform. Today, Facebook has become a lot more of a controlled environment, which means that many developers have been forced away, but many others have gone on to produce really impressive applications. With the introduction of Games and Applications dashboards alongside a unified stream social application, developers have to focus a lot more of their attention on users' experiences.

Add Application Tab FBML Element

Like the deprecated `<fb:add-section-button>` FBML element, Facebook intends to create a related element for adding an application tab. However, at the time of this writing, no information was available as to its name or related attributes.

Facebook officially deprecated boxes and application info sections, which left application tabs as the only way for users to showcase their favorite applications on their profile. There are still modifications being made to how application tabs will be rendered, but the information in this section should give you enough information to start implementing them in conjunction with your Facebook applications. The deprecation has meant that many methods have been removed from the API, including the following:

- `profile.getFBML`
- `profile.getInfo`
- `profile.getInfoOptions`
- `profile.setFBML`
- `profile.setInfo`
- `profile.setInfoOptions`

If you are a new Facebook developer, the changes will mean that you now only have a single integration point to worry about. For developers who have been working with the platform for a longer period of time, these changes have been met with some negativity. Ultimately, however, these should improve the platform. They also allow you to focus more on users' experience of your applications and will be replaced by newer features as time goes by.

Configuring and Installing an Application Tab

Application tabs are displayed within Facebook next to a user's Wall, Info, and Photos tabs, and must be added explicitly by the user. An application tab is currently 520 pixels wide and can be used to render information pulled directly from your application servers as either an `<iframe>` or FBML. Other features of application tabs are that they can be used to load AJAX but cannot autoplay Adobe Flash, `onload` JavaScript, or use `<iframes>`. When interacting with an application tab on a friend's profile, a user's identifier is passed within an `$facebook->fb_params["user"]` parameter alongside the owner's identifier, which is passed within an `$facebook->fb_params["profile_user"]` parameter. An example of how these two parameters can be used is shown in the next section.

Other Canvas Settings

A number of other canvas settings are available within the Canvas tab that are not used within this book but are essential if you want to create an internal Facebook application. The default setting for Facebook Platform website applications is an `<iframe>` render, which means that any standard page will be wrapped within a Facebook frame and displayed to the viewer. For example, if you set the canvas callback URL to the location where you uploaded your files from Chapters 5–7, you will be presented with your `index.php` page.

Because application tabs are used within the Facebook environment, their location must be set relatively to a canvas page URL. And because Facebook Platform website integration has been the focus of this book, a canvas page URL has not yet been set. We can rectify this by navigating to the "Canvas" tab of your application's settings and by providing a unique base URL prefixed by http://apps.facebook.com/. You should also set a canvas callback URL, which is the file or directory on your web server that will be served by Facebook as content for internal canvas pages. For example, if you set your canvas page URL to http://apps.facebook.com/myfacebookapp/ and your canvas callback URL to http://myfacebookapp.com/canvas/, that means that if a user visits http://apps.facebook.com/myfacebookapp/foo.php, it will be rendered from http://myfacebookapp.com/canvas/foo.php. Before continuing, check that the render method on the "Canvas" tab is set to "IFrame" because the Facebook Platform library will be used in this section.

Modifying Your `config.php` File

The `config.php` file that was used in Chapters 5, 6, and 7 should be updated with two new parameters called `CANVAS_PAGE_URL` and `CANVAS_CALLBACK_URL`. These should be inserted as with the other parameters within that file and without their trailing forward slash (/).

For this chapter, you should create a new directory called `canvas` within your existing file structure from Chapters 5, 6, and 7, and upload two files, `index.php` and `tab.php`, along with an `xd_receiver.htm` file. Ensure that the references to the Facebook API PHP client library in `index.php` and `tab.php` are relative to your existing directory structure. The code in Listing 8.1 demonstrates a sample Facebook canvas page showing a simple greeting along with a user's identifier and name.

Listing 8.1 **The `index.php` File Demonstrating a Simple Facebook Canvas Page**

```
1  <?php
2  include "../config.php";
3  include "../functions.php";
4  include "../facebook-platform/php/facebook.php";
5  $facebook = new Facebook(API_KEY, SECRET);
6  $user = $facebook->get_loggedin_user();
7  ?>
8  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
9  <html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:fb="http://www.facebook.com/2008/fbml">
10 <head>
11  <title>Test Tube</title>
12 </head>
13 <body>
14  <h1>Canvas Page - Test Tube</h1>
15  <?php echo "<p>User Identifier: ".($user ? $user : "Unknown").
    "</p>"; ?>
16  <?php echo '<p>Facebook Name: <fb:name uid="'.$user.'"
    useyou="false"></fb:name></p>'; ?>
17  <script src="http://static.ak.connect.facebook.com/js/api_lib/
    v0.4/FeatureLoader.js.php" type="text/javascript"></script>
18  <script type="text/javascript">
19   FB.init("<?php echo API_KEY; ?>", "xd_receiver.htm");
20  </script>
21 </body>
22 </html>
```

This basic page will be rendered inside an `<iframe>`, which means that the Facebook PHP client library alongside the client-side Facebook Platform library will be utilized. The PHP library and configuration files are included on lines 2 to 5, and the current user is assigned on line 6. Because any Facebook user can view this page, it might be that you do not have a `$user` available. Therefore, this must be tested on line 15. To require that a user has logged in when visiting your canvas page, you add `$facebook->require_login();` before the call on line 6. The Facebook Platform library is included on line 17 and initialized on line 19, referencing the recently uploaded `xd_receiver.htm` file. Save the code in Listing 8.1 as `index.php` and upload it to your `canvas` directory, which should be set as your canvas callback URL. If you visit your canvas page URL, you should be presented with a page similar to that shown in Figure 8.3.

Unlike pages within a canvas, which can be an `<iframe>`, your `tab.php` file must be rendered as valid FBML, which is demonstrated by the following code by wrapping content within two `<fb:fbml>` tags:

```
<fb:fbml>
 <h1>Tab Page - Test Tube</h1>
 <p>Hello, World!</p>
</fb:fbml>
```

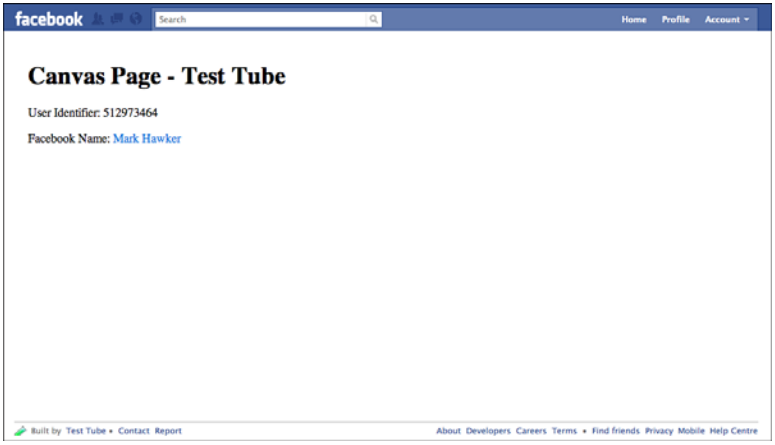

Figure 8.3 Example canvas page for the Test Tube application.

The `<fb:fbml>` tag has an optional `version` parameter that, if omitted, will render the content in the latest version of FBML. To view this number, you can use the `<fb:fbmlversion />` to render the version number within your application. After you've uploaded the `tab.php`, go back into your application's settings and enter a tab name and tab URL on the Profiles tab. In this instance, the tab URL should be set to `tab.php` to mirror the file you have just uploaded, and the tab name should be set appropriately. Once you have saved your settings, visit your own Facebook profile and, if you have installed your application, you should be able to click the plus sign (+) next to the Wall, Info, and Photos tabs and select the tab you just created, as illustrated in Figure 8.4.

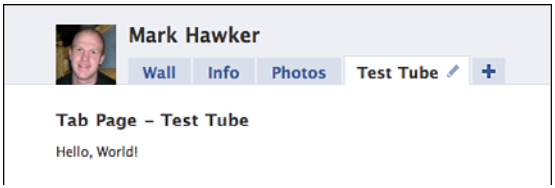

Figure 8.4 Example application tab for the Test Tube application.

Up until now, the application tab contains only static content, and so the next section looks at how these tabs can be extended to add more personalized information tailored to its owner and viewers.

Extending an Application Tab

Before adding additional functionality, it is worth evaluating which Facebook parameters are contained within an application tab, both when viewing a friend's profile and when interacting with it (such as sending a message). When you are viewing a canvas page, the following parameters are exposed and can be accessed by using the `$facebook->fb_params` array:

- **`in_canvas`, `added`, `in_profile_tab`, and `in_new_facebook`**
  These parameters should all be set to a `1`, indicating that the profile owner has added the application and that the viewer is located within the "Profile" tab. The `in_new_facebook` parameter is used for legacy reasons when Facebook was transitioning between old and new layouts. If the `in_profile_tab` is not set to `1`, you should code in functionality to redirect the user to your application's canvas page or display an error message.
- **`friends`, `locale`, `profile_update_time`, `profile_user`, `profile_id`, and `ext_perms`**
  These parameters are associated with the profile owner and contain a comma-separated list of their friends alongside their identifiers and any extended permissions they have granted the host application.
- **`request_method`, `time`, `expires`, `profile_session_key`, `api_key`, and `app_id`**
  The final parameters are used when handling Facebook actions that require sessions, such as extracting the profile owner's friends. When you are using the official client libraries, parameters such as `api_key` are less important because the library handles much of its complexity for you.

These parameters are also accompanied by a signature that can be accessed by using `$_POST["fb_sig"]`. All the parameters above are made available whether the viewer has added the application or not. However, if users intend to interact with the application (for example, submitting a form) but they have not added your application, only the following parameters will be exposed within the `$facebook->fb_params` array: `profile`; `locale`; `in_new_facebook`; `sig_time`; `added`, which will be set to `0`; `api_key`; and `app_id`. In the instance, the identity of the viewer is not accessible to your application. If the viewer has added your application, this will expose the following additional parameters:

- `profile_update_time`, `expires`, `session_key`, and `ext_perms`, which were detailed earlier, although the `session_key` is linked with the profile viewer and not the owner.
- The viewer identifier is now also made available via the `user` parameter.

One of the great features about application tabs is their capability to utilize Mock AJAX calls to perform dynamic actions or to submit forms inline without having to redirect the user. In the remainder of this section, you learn how to create an application tab that enables viewers to leave a basic text comment for their friend to view. Figure 8.5 gives an example of what the final page should look like. It consists of a form that handles submissions using Mock AJAX, a comments box, and functionality to prompt users who have not already authorized your application to add it and grant extended permissions to write data to their stream.

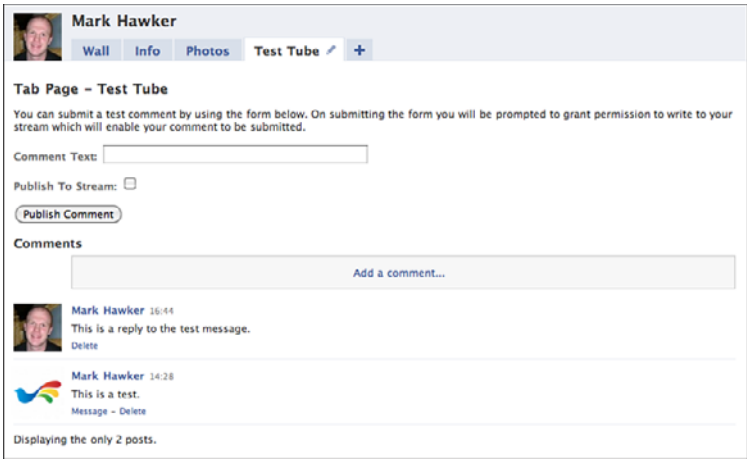

Figure 8.5 Extended application tab for the Test Tube application.

To create the application tab shown in Figure 8.5, you must amend `tab.php` and create a new file for handling the comment submissions called `post.php`. The first step is to create the skeleton of the application tab, which will include the Facebook API PHP client library. This library will be used to validate all parameters and to ensure that the user is, in fact, viewing from within Facebook. Because the canvas callback URL exists on your own web server, it is possible for users to type that file location into their web browser outside of Facebook, which means that they must be redirected back to Facebook to prevent any malicious access. This can be achieved by adding the code in Listing 8.2 to `tab.php`.

Listing 8.2 **Example for the `tab.php` File Demonstrating a Simple Application Tab**

```
1  <?php
2  include "../config.php";
3  include "../functions.php";
4  include "../facebook-platform/php/facebook.php";
5  $facebook = new Facebook(API_KEY, SECRET, $_POST["fb_sig_profile_
   session_key"]);
```

```
6  $facebook->require_frame();
7  $facebook_parameters = $facebook->get_valid_fb_params($_POST, null,
   "fb_sig");
8  $profile_user = $facebook_parameters["profile_user"];
9  if($facebook_parameters["in_profile_tab"] == 1) {
10 ?>
11 <fb:fbml>
12  // Your Application Logic Goes Here
13 </fb:fbml>
14 <?php
15 } else {
16  // Either redirect the user by setting $facebook->redirect(CANVAS_
    PAGE_URL); or by presenting them with a warning saying that they are
    not within an application tab.
17 }
18 ?>
```

On line 6, a function has been added that ensures that if users type in the URL to your tab on your domain in their web browser they will be redirected back to a Facebook-hosted page. Before you add any application logic, it is worth drafting out what use your new application tab should be able to handle. Because it will have been added by a user, it is known that the user has already authorized your application, and it is possible that you can use the user identifier or other details to extract data that you hold about that user from your database and display that on your tab. Because you are provided with a session key, a number of functions can be performed on the tab itself (for example, extracting the user's friends, photos, or events). However, you will not be able to publish to the user's stream or retrieve any protected data on the user's behalf. When a user's friends access your application tab, they will be in a "passive" mode. This means you cannot access their user identifier, and so cannot determine whether they have authorized your application. Facebook provides two mechanisms for handling this issue:

- Adding a `requirelogin="true"` parameter to all links. This will pop up a Facebook dialog box so that users can authorize your application before proceeding if they have not already.
- When using Mock AJAX, you can add an `ajax.requireLogin=1` parameter so that if viewers submit your comment form and they are not a user of your application, they will be prompted to authorize first before the comment is posted.

Both mechanisms should arrive at the same results. However, because you'll be learning about Mock AJAX in this example, the second option is used. When submitting their comment, viewers can also select whether they want to post their comment to their stream via the `comments.add` method. Note that you could post the response as a stream

attachment or even via the Dashboard API (described earlier in this chapter). The `comments.add` method is used for convenience because a comments box can be placed on the application tab to show feedback to the user. Posting to their stream also requires that a viewer has granted the `publish_stream` extended permission, which will be prompted by the `Facebook.showPermissionDialog()` JavaScript function.

The comments form can be constructed using the following code, which you should use in place of the comment on line 12 of Listing 8.2:

```
1  <h1>Tab Page - Test Tube</h1>
2  <p id="comment_response">You can submit a test comment by using the
   form below. On submitting the form you will be prompted to grant
   permission to write to your stream which will enable your comment to
   be submitted.</p>
3  <form>
4   <p><label for="comment_text">Comment Text: </label><input type="text"
    name="comment_text" id="comment_text" value="" size="50"
    maxlength="140" /></p>
5   <p><label for="publish_comment">Publish To Stream: </label><input
    type="checkbox" name="publish_comment" id="publish_comment" /></p>
6   <p><input type="submit" value="Publish Comment" onclick="
    submit_form('comment_response'); return false;" onsumbit="return
    false;" /></p>
7  </form>
8  <h2>Comments</h2>
9  <div id="comments_box">
10  <fb:comments xid="c_<?php echo $profile_user; ?>" canpost="true"
    candelete="true"></fb:comments>
11 </div>
```

The code above displays a simple prompt to the user on line 2. This prompt will be replaced when the form is submitted by either a success or error response. The form itself is defined in lines 3 to 7. It does not include traditional `action` and `method` attributes because you will be using the `onclick` action of the Submit button to post a comment. Form elements include a mixture of `name` and `id` attributes because their values and states need to be evaluated for validation and submission. The comments box on line 10 is wrapped inside a `<div>` because when a user submits a comment there is no way of "refreshing" its contents without refreshing the application tab. The `xid` of the `<fb:comments>` FBML element is set to that of the profile owner and is prefixed by a `c_`, because Facebook sometimes has issues displaying comments boxes that are purely numeric.

The next element that needs to be created is the JavaScript function `submit_form()`, which includes the `id` of the element to update after submission. The JavaScript for this example is split into two parts. The first detects whether comment text was added and whether viewers have chosen to publish their comment to their stream. If both are true, they are presented with a Permissions dialog box to grant extended permissions. If permissions are granted, a successful callback will be triggered, and the comment will be

posted to their stream. If denied, the comment will still be posted but will not appear in their stream. The second part is the Mock AJAX itself, which is used to submit the comment and update the user interface. The `submit_form()` function looks like this and should be placed inside a `<script type="text/javascript">` element:

```
1  function submit_form(form) {
2   comment_text = document.getElementById("comment_text").getValue();
3   if(!comment_text == "") {
4    publish_comment = document.getElementById("publish_comment").
     getChecked();
5    if(publish_comment) {
6     Facebook.showPermissionDialog(
7      "publish_stream",
8      function(response) {
9       if(response) { do_ajax(form, publish_comment); }
10      else {
11       do_ajax(form, false);
12       document.getElementById("publish_comment").setChecked(false);
13      }
14     }
15    );
16   } else {
17    do_ajax(form, false);
18   }
19  } else {
20   document.getElementById("comment_text").setStyle({color: "white",
     background: "red"});
21  }
22 }
```

Facebook's implementation of JavaScript, FBJS, is slightly different to JavaScript in handling variable names. In all instances, variables are prefixed by your application ID, which creates a more controlled and sandboxed environment that prevents malicious screen refreshes and other potentially dangerous scripting abilities. Some useful FBJS commands are shown on line 2 for getting the value of a text box, on lines 4 and 12 for getting and setting the state of a check box, and on line 20 for setting the style of a text field. Further details are available in the next section for how to add event listeners and other advanced functionalities to your application tab. The `Facebook.showPermissionDialog()` function on lines 6 to 15 is broken down as follows:

- Line 7 defines the extended permission or permissions that are being requested. In this instance, you require only the `publish_stream` permission, but multiple permissions can be requested by supplying a string of comma-separated values.
- Lines 8 to 14 are the `callback` function, which is invoked if the user allows the permission that leads to the call on line 9. If the user denies permission or closes the Permissions dialog box, the response will be `null`. This will still submit the

comment but will ensure that it does not attempt to publish to their stream. Because this function is being called as a result of the user checking the Publish Comment check box and then being denied, the check box is set to "unchecked" to improve user experience should the user attempt to submit again. Both callback paths will call a `do_ajax()` function (detailed below).

The remainder of the `submit_form()` function is to handle if users do not want to publish to their stream. Under this scenario, the `do_ajax()` function is called, much like if they deny the `publish_stream` extended permission. If they do not provide any comment text, the background of the text field will be set to red and the text to white. The `do_ajax()` functions should be placed below `submit_form()` and contains the following code:

```
1  function do_ajax(div, publish_comment) {
2   comment_text = document.getElementById("comment_text").getValue();
3   if(!comment_text == "") {
4    var ajax = new Ajax();
5    ajax.responseType = Ajax.JSON;
6    ajax.ondone = function(data) {
7     document.getElementById(div).setInnerFBML(data.fbml_response);
8     document.getElementById("comments_box").
      setInnerFBML(data.fbml_comments);
9     document.getElementById("comment_text").setValue("");
10    document.getElementById("comment_text").setStyle({
       color: "black", background: "white"
      });
11   }
12   ajax.onerror = function() {
13    document.getElementById(div).setInnerFBML('<fb:error message="There
      was an error submitting the form." />');
14   }
15   var params = {
16    "comment_text": comment_text,
17    "owner": <?php echo $profile_user; ?>,
18    "publish_comment": publish_comment
19   };
20   ajax.requireLogin = 1;
21   ajax.post("<?php echo CANVAS_CALLBACK_URL; ?>/post.php", params);
22  }
23 }
```

As with the `submit_form()` function, the `do_ajax()` function first tests to see that comment text has been entered. If it hasn't been, it will not submit any data to Facebook. On line 4, an AJAX object is created, and its `responseType` is set on line 5. The `responseType` can be set to `Ajax.JSON`, `Ajax.RAW`, or `Ajax.FBML`, which dictates the format in which the AJAX object expects data to be returned. The most flexible format is

Ajax.JSON, which will be demonstrated in the example in this chapter. Lines 7 and 8 use two JSON strings, fbml_reponse and fbml_comments, which will become clear after exploring the server-side file generating the response. There are two cases for AJAX requests, which are ajax.ondone and ajax.onerror for handling successful or other responses. The ajax.ondone function on lines 6 to 11 is used to update the comments_box and for resetting the Comments text field to its original state. The final part of the function is shown on lines 15 to 21, which are used to set up POST parameters, comment_text, owner, and publish_comment, to require that users have authorized the application and to actually post the data.

The <fb:js-string> FBML Element

When setting the innerFBML of an element, you might find that Facebook refuses to add the content that you specify. The <fb:js-string> FBML element is provided specifically for this case—another is for Facebook Dialogs—and contains a single var parameter, which is the name that it will be referenced by and will contain the FBML that you want to be added. The <fb:js-string> should be placed within an <fb:fbml> element and will not be displayed to users. The var should be passed as the single parameter to an innerFBML() function.

Your post.php is used to perform specific server-side Facebook functions and to return the response back to the do_ajax() function. The CANVAS_CALLBACK_URL parameter that was set within the config.php should include the canvas directory to ensure that the post.php file can be found. Listing 8.3 defines an example post.php file. This should be uploaded to your web server alongside tab.php and index.php.

Listing 8.3 **Example post.php File Demonstrating Adding a Comment and Returning Data Back to an Application Tab**

```
1  <?php
2  include "../config.php";
3  include "../functions.php";
4  include "../facebook-platform/php/facebook.php";
5  $facebook = new Facebook(API_KEY, SECRET);
6  $facebook_parameters = $facebook->get_valid_fb_params($_POST,
   null, "fb_sig");
7  if(empty($facebook_parameters)) {
8   $facebook->redirect(CANVAS_PAGE_URL);
9   exit;
10 }
11 if($facebook_parameters["is_ajax"] == 1) {
12  $owner = $_POST["owner"];
13 } else {
14  $owner = $facebook_parameters["profile"];
15 }
```

```
16 $viewer = $facebook_parameters["user"];
17 $comment_text = $_POST["comment_text"];
18 $publish_comment = $_POST["publish_comment"];
19 $facebook->set_user($viewer, $facebook_parameters["session_key"]);
20 $json = array();
21 $json["fbml_comments"] = '<p>The page <a href="http://www.facebook.com/
   profile.php?id='.$owner.'&v=app_'.$facebook_parameters["app_id"].'">
   must be refreshed</a> to view recently-submitted comments.</p>';
22 try {
23  $title = "Test Tube";
24  $url = CANVAS_PAGE_URL;
25  $comment = $facebook->api_client->comments_add("c_".$owner,
    $comment_text, $viewer, $title, $url, $publish_comment);
26  $json["fbml_response"] = '<fb:success message="Your comment was added
    and will be viewable the next time you visit this tab." />';
27 }
28 catch(Exception $e) {
29  $json["fbml_response"] = '<fb:error message="'.$e->getMessage().
    '" />';
30 }
31 echo json_encode($json);
32 ?>
```

As with the `tab.php` file, you must cater for the fact that your `post.php` file will be accessed externally, which is the reason for including lines 7 to 10. Because Mock AJAX is being used, Facebook adds another parameter called `is_ajax` but does not pass the profile parameter, which is why the owner `POST` parameter was set within the `do_ajax()` function. Other parameters are set on lines 16 to 18, and then the profile viewer is set as the active user on line 19. An empty array is created on line 20, which is finally converted to a JSON string on line 31 and which is returned to `do_ajax()`. As an example, line 21 is the text that replaces the initial `comments_box` container and is accessed within `do_ajax()` using `data.fbml_comments`. If you want to return data that is to be set using `setInnerFBML`, it must be prefixed with `fbml_` within the `$json` parameter. The `comments.add` method is called on line 25 using the `comment_text`, and the final parameter dictates whether the comment is published to the viewer's stream.

After you have created the `post.php` file, you should upload it to your web server, and you should be ready to test out your new application tab. From here, you could try out another publishing method such as `stream.publish` or add additional functionality such as listing the owner's friends who have commented or displaying richer comments that include images. The final section looks at how to use the FBJS, and in particular the Animation library, which can be used to create "tweening" CSS fading background colors and styles, to hide and show block-level elements, and to ease animations for smoother transitions.

Dynamic Content and the Facebook JavaScript (FBJS) Library

The Facebook JavaScript (FBJS) library is a solution prepared by Facebook to enable developers to execute JavaScript within their applications. Because allowing developers to perform the full range of JavaScript commands could lead to malicious use, FBJS attempts to provide a happy medium for providing access to simple animations and to utilizing event listeners and implementing Facebook dialog boxes. As you may have seen if you have tried to use JavaScript within Facebook before, all your variable names and functions are prefixed with an application ID. If your application ID is `1234567890` and you have a function named `foo()`, it becomes `a1234567890_foo()`. In the code for the application tab in the previous section, it was not possible to simply refresh the tab using `window.location.reload()` because of this, although you could use `document.setLocation()`, which is provided in the FBJS library. Because application tabs are the only way of enabling a user to showcase your application, it is important to add features such as Mock AJAX and animations to improve the usability of your work and to distinguish yourself from others.

> **Including JavaScript Files**
>
> If you have a rather large JavaScript file, you can use a `<script>` tag and set the `src` to include the remote file. As Facebook caches the file to reduce the burden on your own servers, you should suffix your files with a version number after each major update (for example, `foo.js?v=0.1`) to ensure that Facebook caches the new file.

The FBML Test Console (http://developers.facebook.com/tools.php?fbml) is a great resource for testing out your FBJS before deploying to an application tab (see Figure 8.6). It can also be used to test out a Facebook Platform application or to trial Facebook API methods before production.

You can set the Position drop-down menu to `tab` to ensure that the correct proportions are being shown onscreen. When previewing your application in the Test Console, you are presented with a preview of how your application tab will look, the contents of the HTML that Facebook will generate, and a simple list of errors (as well as the ability to view a profile from the perspective of another user by setting the Profile text field). The remainder of this section uses the FBML Test Console to experiment with the various features of the FBJS library.

Facebook Animation Library

Facebook provides an easy-to-use library for creating a richer user interface for your users via CSS both inside Facebook and outside through an animation library (http://developers.facebook.com/animation/). This library could therefore be used to create animations for other applications that are not Facebook driven but utilize basic animations such as

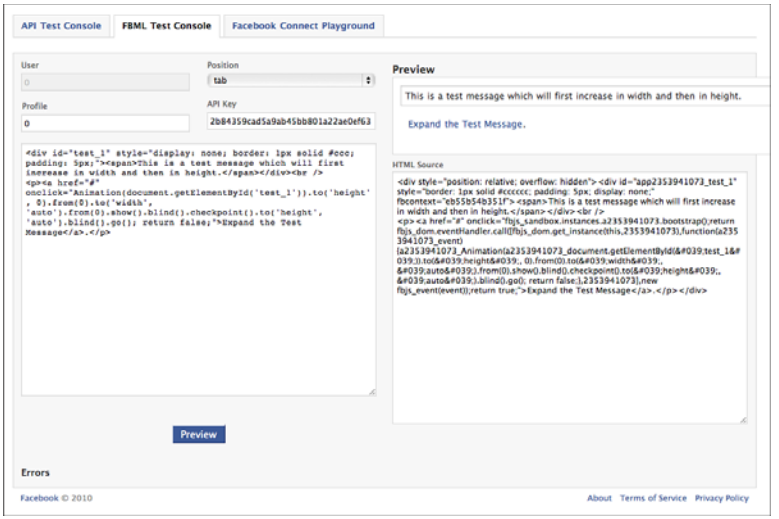

Figure 8.6 Screen shot of the FBML Test Console showing an example application tab.

creating shading effects that "tween" between background or text colors or hiding and showing page elements. These could be used to animate a particular element and can be achieved in the following ways by populating the `onclick()` parameter of any element:

- **`Animation(this).to("background', "#000").go();`**

  This function will transition the element's current `background` color to black (`#000`) and is "executed" by supplying the final `.go()` method. The use of this ensures that the animation is performed on the current element but any other DOM object could also be passed into this function for manipulating elements in other areas of a page.

- **`Animation(this).to("background", "#f00").to("color", "#fff").go();`**

  You can string multiple styles together, such as `background` and `color`, as shown in the example. Both transitions will run smoothly in parallel, which means that as the background is changing color, so will the color of the text.

- **`Animation(this).to("background", "#fff").from("#000"). go();`**

  To transition between two styles irrespective of the current style, you can use a `.from()` method. In this instance, this meant changing the `background` from white (`#fff`) to black (`#000`).

- **`Animation(this).by("font-size", "1px").go();`**

  The `.by()` method can be used to increment or decrement an attribute, such as `font-size`, `width`, `height`, or `left` or `right` positioning.

- **`Animation(element).to("height", 0).to("opacity", 0).blind().hide().go();`**

  By setting the `height` and `opacity` of a supplied `element`, you can automatically hide it from view. You should also set the element's `overflow` style to `hidden`, which will prevent images contained within the `element` from still being shown despite it having no size. The `.blind()` method is used to prevent automatic text wrapping from occurring while the `element` is being resized.

- **`Animation(element).to("height", "auto").from(0). to("width", "auto").from(0).to("opacity", 1).from(0).blind().show().ease(Animation.ease.end).go();`**

  Revealing elements that have a `display` style set to `none` works in a similar way to hiding them but requires both a `.to()` and `.from()` method as well as `.show()` in replace of `.hide()`. A final, `.ease()` method was added to the animation, which will mean the element will "ease" into being revealed. Other options are `Animation.ease.begin` and `Animation.ease.both`, which will start slow and end fast or start and end slow, respectively.

All the animations above will occur over a duration of 1,000 milliseconds (1 second), but you can add a `.duration()` method right before `.go()` should you want the animation to last a longer or shorter time. The code examples available for this chapter contain a few animations to demonstrate how they function on application tabs and how they could be implemented in your own applications. A final advanced feature of the Animation library is checkpoints. Checkpoints are useful if you want to build an animation that consists of two or more logical steps that are part of a single animation. Example could be first increasing a width and then increasing its height or increasing the size of an element and then changing its color. This can be demonstrated using a simple example:

```
<div id="test_1" style="display: none; border: 1px solid #ccc; padding:
5px;">
 <span>This is a test message which will first increase in width and
 then in height.</span>
</div>
<a href="#" onclick="Animation(document.getElementById('test_1')).
to('height', 0).from(0).to('width', 'auto').from(0).show().blind().
checkpoint().to('height', 'auto').blind().go(); return false;">Click
to Expand</a>.
```

It is also possible to "stagger" checkpoints so that an action can be executed midway through the first animation. To implement this feature, you can add an additional parameter to the `.checkpoint()` function, which must be a number that ranges from `0` to `1`, where `0` will not render the animation at all and a value of `1` will render the animation straight after the first has finished. For example, in the code above, you could set the checkpoint to `0.5` to start growing the `height` of the element halfway through its `width` increase. This can also be accompanied by a `.duration(500)` function just before `.go()`

to ensure that both animations finish at the same time. A trick to delay animations is to use the following:

```
Animation(element).duration(3000).checkpoint().to("width", "auto").go();
```

This code would pause for 3,000 milliseconds (3 seconds) and then adjust the width of the given `element`. A use case for this may be to present a message after a certain period of time to the user or to hide a message after a number of seconds has elapsed. The final advanced feature of checkpoints is to use callbacks within the `.checkpoint()` function for performing animations on other elements as well as the current `element`. This can be achieved by using `.checkpoint(1, function() { Animation(...); })` and nesting your animation within the two parentheses. Remember that you can also save these animation chains as functions and thus greatly reduce the amount of code you are typing and make it more readable if you call functions such as `expand()`, `contract()` or `growThenFadeToBlack()`.

Facebook Dialogs

> **The `<fb:dialog>` FBML Element**
>
> Facebook has a beta version of an `<fb:dialog>` element that is a condensed version of the FBJS equivalent discussed in this chapter. The element can be invoked by adding a `clicktoshowdialog` attribute to any element. It is recommended that you use the FBJS version until Facebook confirms the `<fb:dialog>` element, which is expected in mid-2010.

Facebook uses dialog boxes to alert users of messages that they have deleted and to alert them about errors and many other scenarios. To make your application blend in with their environment, they provide a `Dialog` object that can be manipulated to show a pop-up message called `Dialog.DIALOG_POP` or a contextual message called `Dialog.DIALOG_CONTEXTUAL`, which displays an inline dialog box rather than a pop-up. Both types of dialog work in similar ways, except that the contextual dialog can be displayed close to where the user's cursor is pointing or around a certain element. A simple dialog box can be created by using the following code:

```
<p><a href="#" onclick="new Dialog(Dialog.DIALOG_POP).showMessage('Test
Dialog Box', 'Hello, World!', 'Close'); return false;">Click to Test
Dialog Box</a></p>
```

The dialog box shows a message which has the title `Test Dialog Box`, the content set to `Hello, World!`, and its only button set to `Close`. The `.showMessage()` function could be replaced by `.showChoice()`, which accepts an additional parameter for allowing a cancel option. A more thorough example of using dialogs is to evaluate which action the user has chosen and to update an element:

```
1  <p>Do you like social programming? <a href="#" onclick="confirm('Do
   you like social programming?', this);">Click to Answer</a></p>
2  <p id="response">Unknown Response</p>
3  <script type="text/javascript">
4  <!--
```

```
5  function confirm(text, context) {
6   var dialog = new Dialog(Dialog.DIALOG_CONTEXTUAL);
7   dialog.setContext(context).showChoice("Social Programming", text,
    "Yes", "No");
8   dialog.onconfirm = function() {
9    document.getElementById("response").setTextValue("Yes, I do.");
10  };
11  dialog.oncancel = function() {
12   document.getElementById("response").setTextValue("No, I don't.");
13  };
14  return false;
15 }
16 //-->
17 </script>
```

In this example, the results of the dialog box lead to the `response` element being updated either on being confirmed (lines 8 to 10) or canceled (lines 11 to 13). You can also see how the `.setContext()` function was used to ensure the dialog appeared close to the `Click to Answer` text. The final example of dialogs makes use of the `<fb:js-string>` FBML element to show a rich select box to the user within a message and enables them to update a string of text based on the color that they select:

```
<p id="body_text">This is some standard text.</p>
<p><a href="#" onclick="update_text_color();">Update Text Color</a></p>
<fb:js-string var="color_picker">
 <p><b>What is your favorite color?</b></p>
 <p>
  <select id="color_select">
   <option value="black">Default</option>
   <option value="red">Red</option>
   <option value="green">Green</option>
   <option value="pink">Pink</option>
  </select>
 </p>
</fb:js-string>
<script type="text/javascript">
<!--
function update_text_color() {
 var dialog = new Dialog(Dialoh.DIALOG_POP).showChoice("Color Picker",
 color_picker, "Pick", "Cancel");
 dialog.onconfirm = function() {
  var color_text = document.getElementById("color_select").getValue();
  document.getElementById("body_text").setStyle({color: color_text});
 };
 return false;
}
//-->
</script>
```

The main difference in this example is that instead of passing a string of text into the `.showChoice()` function, the `var` of the `<fb:js-string>` element is used. This method can prove particularly effective if you intend to create a rich form that the user has to fill out or if you intend to include multimedia in your dialog box. The only methods that have not been explored are `.hide()`, which can be used to hide a dialog box if it is already opened (such as if you intend to open multiple dialog boxes or ensure that they are all properly closed), and `.setStyle()`, which can add styling to the dialog box.

Handling Events with an Event Listener

You might sometimes want to detect whether users have clicked an element on your application tab or moved their mouse over a text field or image. In these instances, you can set up an event listener that sits in the background of your code waiting for actions to occur. Facebook provides its own facilities to "listen" for events and has thus extended the W3C `addEventListener()` method. Event listeners are broken into three components:

- A string related to the event type that is being listened for, which includes mouse events such `click`, `mousedown`, `mouseup`, `mouseover`, `mousemove`, `mouseout`, or keyboard events like `keyup`, `keydown`, or `keypress`. To detect a particular key press, you can use the `keyCode` property of an `Event` object to perform specific functions dependent on keys. You can also use the `Event` object to detect whether the `ctrlKey`, `shiftKey`, or `metaKey` were pressed.
- A callback function that handles the event and triggers whatever functionality you want to implement. This could be updating a text box, adding text to row tables, or performing search "typeahead" functions. Two important functions can be set within this function: `stopPropagation()`, for preventing the listener from being added to any parent elements; and `preventDefault()`, for stopping an element's "normal" behavior (such as preventing clicking a link from directing the user). In the case of a link, you must also set its `onclick` attribute to `return false;`.
- The final parameter must be set and relates to a `useCapture` behavior, which should be set to `false`. This will prevent events being triggered for descendants of the element that triggers that particular listener.

You can use event listeners in two ways depending on what types of actions you want to capture. The first type of listener is used to encompass multiple elements and handle their logic within the callback function. For example, suppose you have a catalog of images and you want to update a text box to describe the image based on what product the user has rolled his mouse cursor over. You can do so using the following code:

```
<p id="product_description">Roll your mouse over an image to update this
description.</p>
<div id="products">
 <p id="image_1"><img src="..." ... /></p>
 <p id="image_2"><img src="..." ... /></p>
</div>
```

```
<script type="text/javascript">
<!--
function handler(event) {
 var product_description = document.getElementById("product_description");
 if (event.type == "mouseout") {
  product_description.setTextValue("Roll your mouse over an image to
  update this description.");
  return true;
 }
 var product_id = event.target.getId();
 var product_text = "";
 switch(product) {
  case "image_1":
   product_text = "This is the first product.";
   break;
  case "image_2":
   product_text = "This is the second product.";
   break;
  default:
   product_text = "This is an unknown product.";
 }
 product_description.setTextValue(product_text);
}
document.getElementById("image_1").addEventListener("mouseover", handler);
document.getElementById("image_1").addEventListener("mouseout", handler);
document.getElementById("image_2").addEventListener("mouseover", handler);
document.getElementById("image_2").addEventListener("mouseout", handler);
//-->
</script>
```

The code above would display two images and accompanying text and has two listeners, `mouseover` and `mouseout`, which will either update the `product_description` element with a product description or reset it to its default text. The `event.target.getId()` function ensures that the correct element is identified, and then the JavaScript logic is tailored to that identifier. Another way to add an event listener is to completely separate the code from your application tab content:

```
<div id="test" style="border: 1px solid #ccc; padding: 5px; height: 50px;
width: 100px;" onclick="return false;"></div>
<script type="text/javascript">
<!--
function random_number(low, high) {
 return Math.floor((Math.random() * (high - low)) + low);
}
function color(obj) {
 var red = random_number (0, 255);
 var blue = random_number(0, 255);
```

```
 var green = random_number(0, 255);
 var color = red + ", " + green + ", " + blue;
 obj.setStyle("color", "rgb(" + color + ")");
}
function load() {
 var obj = document.getElementById("test");
 obj.addEventListener("click",
  function(event){
   color(obj);
   event.stopPropagation();
   event.preventDefault();
   return false;
  }, false);
}
load();
//-->
</script>
```

The code above will display a box that is clickable and that will change to a random color generated by the `color()` function. The difference in this example is that a `click` event is being captured and so the `preventDefault()` function is called to prevent the usual action of clicking an object. Note that only this second event listener can be validated using the FBML Test Console and the previous example of the product catalog must be hosted on a live application tab. Event listeners are the final component of the FBJS library explored in this section and can be used in combination with the Animation library and Mock AJAX. You should now feel well enough equipped to create an interactive and dynamic application tab that will keep your users coming back and that will persuade their friends to add one of their own.

Summary

This chapter described how you can use dashboards in your Facebook Platform application through the Dashboard API. Through the Dashboard API, you can post news items to a user's dashboard, promote friends' activities, and utilize activity counters. The second part of this chapter focused on application tabs as a way of sharing your application's information with users and their friends. Following the deprecation of profile boxes, application tabs are the only mechanism for enabling users to personalize their profiles and showcase their favorite applications. You were shown how to configure and install an application tab and how to add Mock AJAX functionality. The final part of this chapter detailed the Facebook JavaScript (FBJS) library, which you can use to add animations, dialogs, and event listeners.

Index

A

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X