Chapter No. 10
"Checkins and Facebook Places"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.10 "Checkins and Facebook Places"

A synopsis of the book’s content

Information on where to buy this book

About the Author

**James Ford** lives and works in the relatively sedate Midlands of England, and is constantly thankful that he doesn’t have to contend with public transport or large volumes of traffic on a regular basis. Occasionally being stuck behind a tractor for several miles is a small price to pay for that!

Macromedia Flash 8 represented his first serious interaction with a programming language, an epiphany whereupon his BA in Design for Interactive Media became far more focused on the Programming side of things than the Art (although it turned out okay).

He considers himself to be fortunate to have encountered the Flash Platform when he did, as during that time between first contact and today, the Flash Platform has evolved at just the right pace to enable him to keep up with the latest shiny new features.

In addition to developing Facebook-integrated Flash Player applications, James has also developed a few AIR applications, built a couple of Apps for iOS and Android devices, helped develop a load of websites, and blogs about all of this on his website: [http://www.psyked.co.uk/](http://www.psyked.co.uk/).

For More Information:

Flash applications are popular and becoming increasingly social. With Flash applications for Facebook you can tap into a potential audience of half a billion existing users, their connections and affiliations, their uploaded images, posts, comments, and more.

The Flash Facebook Cookbook is packed with recipes for the Graph API and FQL, used for reading and writing data as well as interacting with Facebook anonymously or on behalf of an authorized Facebook User.

The topics covered by the recipes in this cookbook include working with News Feeds, uploading Photos, searching for and plotting Places on a Map and much more. The cookbook has recipes ranging from those that work without any authentication with Facebook to those that do, and act on behalf of a user. Packed with recipes that yield practical demonstrations of the Graph API functionality, the Flash Facebook Cookbook is an essential tool for Flash Platform developers.

What This Book Covers

Chapter 1, Getting Started with Flash and Facebook: This chapter deals with downloading the Facebook ActionScript 3 SDK, registering a new Application on Facebook.com, getting your Flash Builder development environment set up, and getting ready to start working with the Facebook APIs.

Chapter 2, Authenticating with Facebook: Our application can't do much unless it's first authenticated with Facebook, so this chapter covers logging in, logging out, and tapping into existing Facebook sessions, for both Flash Player and AIR-based projects.

Chapter 3, Working with Facebook Permissions: Even authenticated, our application can't do much without the correct permissions, so this chapter covers processes, tactics, and strategies for working with and requesting those Extended Permissions for Facebook users.

Chapter 4, Reading and Writing Data with the Graph API: A generalized introduction to working with the core Graph API, including retrieving specific objects, multiple objects, and object connections; searching for data, limiting and paging results; and an introduction to creating and managing data with the Graph API.

Chapter 5, Loading Data with FQL: FQL is the SQL-like syntax for retrieving data from Facebook, and with it we can retrieve far more specific data than we can with the Graph API, including data that is unavailable via the Graph API.

For More Information:
Chapter 6, Facebook News Feeds and Status Updates: In this chapter, we look at loading Facebook News and Profile Feeds, creating new Status updates and Links, and adding custom actions to links to help promote your Facebook application.

Chapter 7, Comments and "Like this": Following on from the previous chapter, we look at loading, creating, and deleting Comments for Graph API objects, and the same for 'Likes', with both Graph API objects and web URLs.

Chapter 8, Working with Photos, Albums, and Tags: In this chapter, we'll load Photos from the Graph API—both photos of the current user, and photos from the current user. We display these images and overlay Facebook tagging information, and use the Flash Player to upload new images on the user's behalf.

Chapter 9, Working with Groups and Events: In this chapter, we retrieve Facebook Group and Event information, locate them on a Map, create new Events, and respond to Event invitations on the user's behalf.

Chapter 10, Checkins with Facebook Places: In this chapter, we download and integrate the Google Maps components and use them to plot Checkins and Facebook Places. We work with the Geolocation capabilities of HTML5, and use them to search for nearby Facebook Places and create new Checkins.

Chapter 11, Bridging the Flash and Application Gap: Finally, we'll look to improve the integration between your application and Facebook—imitate the Facebook.com website interface, use native dialogs to bypass the need for Extended Permissions, and develop a new class to simplify the authentication and permissions management for our application.

For More Information:
In this chapter, we will cover:

- Downloading and integrating the Google Maps API
- Retrieving Checkin information for the current user and their friends
- Finding Facebook Places near a specific location
- Integrating the HTML5 Geolocation capabilities, Maps, and Facebook Places
- Creating new Checkins at an existing Facebook Place

Introduction

In this chapter, we’re going to look at the location-oriented features of the Graph API, namely Places and Checkins.

These two elements are intrinsically linked—Places are objects in the Graph API which link to real-world geolocation coordinates, existing as locations where a user can register their visit, for a fleeting moment, as a Checkin.

Checkins are different than Events, in that they have much more momentary significance—there’s not the same level of forward-planning and formal attendance with a Checkin as there is with an Event. Likewise, Places are unusual in that they don't really have their own fixed object type in the Graph API. They exist, but as sub-objects of Checkin objects.

Checkins to Facebook Places are made primarily through mobile or location-aware devices, such as, but not exclusively, mobile phones or tablet computers. With the Facebook ActionScript 3 SDK, we are able to retrieve and create Checkins on the user's behalf.

For More Information:  
Checkins and Facebook Places

With the Graph API methods available to our Flash Platform applications, we are able to retrieve information about Places and user's Checkins, plot them on a map, and create new Checkins on the user's behalf.

With the recipes in this Chapter being very location-aware, we're going to require some mapping capabilities in our applications.

**Downloading and integrating the Google Maps API**

There are, of course, a wide variety of mapping components available for use in the Flash Player, but we're going to make use of the Google Maps component.

Why use the Google Maps component?

Aside from it being a well-supported component, it's freely available. The API makes it easy to plot locations and add overlays, there are versions for both Flash and Flex applications, and although Bing is technically the official supplier of the map functionality for the Facebook website, Bing doesn't have a Flash Player-oriented component (that I'm aware of).

How to do it...

The Google Maps Flex component source files can be downloaded from its Google Code repository: `http://code.google.com/apis/maps/documentation/flash/`.

For More Information:

Follow the instructions on that page to obtain a Google Maps API Key, and download the SWC file (which currently is `map_flex_1_20.swc`).

1. To use the Google Maps classes in our Flash Builder applications, add that SWC file to your project's `libs` folder or modify the project's `Flex Build Path | Library path` properties to include the SWC.

2. With the SWC file included in our project, we will be able to create new Map components with MXML, like the following:

   ```xml
   <maps:Map height="400" id="map"
   key="[YOUR GOOGLE MAPS_API_KEY]"
   sensor="true" width="400" />
   ```

   Of course, we should replace the `[YOUR GOOGLE_MAPS_API_KEY]` key in the preceding code with the one that we receive after completing the sign-up process. If we don't, or if we use an invalid API Key for the application's domain, we'll get an error message instead of the Map.

3. Finally, if Flash Builder doesn't add the Google Maps XML namespace automatically to our application's root MXML tag, add the following path `com.google.maps.*`:

   ```xml
   <s:Application height="600" width="758"
   xmlns:maps="com.google.maps.*"
   xmlns:mx="library://ns.adobe.com/flex/mx"
   xmlns:s="library://ns.adobe.com/flex/spark">
   ```

---

**Retrieving Checkin information for the current user and their friends**

In this recipe, we're going to retrieve Checkins information for a single specific user, who will be selected from a `DropDownList` component containing a list of the current user and their friends. Once we have retrieved that Checkins information, which comes as a Graph API connection, we will plot the location of those Checkin on our map component.

---

For More Information:

By the end of this recipe, we should have an application which looks a little like this:

![Select a user to load Checkin information for:](image)

### Getting ready

The /PROFILE_ID/checkins connection contains protected information, so our application will need to be able to authenticate with the Facebook API as a user, and will require the `user_checkins` and `friend_checkins` Extended Permissions.

To simplify the authentication process, the base class of our application should be the `ApplicationBase` class, detailed in the recipe *Creating the ApplicationBase class*, in *Chapter 11, Bridging the Flash and Application Gap*.

The visible structure of our application will contain three main components: a `DropDownList` component with the ID `users_list`, a `Map` component, with an ID of `map`, and a `Label` component, which for the sake of brevity we'll just use with Flex's data binding (no need to give it a specific ID).

---

For More Information:

1. The first step for our application (which is not directly related to this recipe) is to retrieve details of the current user's friends, which we display in the users_list component.

To retrieve and parse that information into an ArrayCollection instance, we would use the following ActionScript code:

```actionscript
Bindable
Var friendsArrayCollection:ArrayCollection;

override public function facebookReady():void {
    Facebook.api("me/friends", friendsResponseHandler);
}

function friendsResponseHandler(success:Object, fail:Object):void {
    friendsArrayCollection = new ArrayCollection();
    var results:Array = success as Array;
    if (success && results) {
        friendsArrayCollection.addItem(currentUser);
        for (var i:int = 0; i < results.length; i++) {
            var user:FacebookUser = FacebookUser.
                fromJSON(results[i]);
            friendsArrayCollection.addItem(user);
        }
        users_list.selectedIndex = 0;
        users_list.dispatchEvent(new IndexChangeEvent(IndexChangeEvent.CHANGE, false, false, -1, 0));
    }
}
```

2. And to display that information in our users_list component, we would add the following MXML properties:

```
<s:DropDownList id="users_list"
    dataProvider="{friendsArrayCollection}"
    labelField="name" />
```

**How to do it...**

Our application interface now includes a DropDownList component populated with a list of the user's friends, and our Google Maps component.

1. With a DropDownList of viable users ready to be selected, we add a selection change event handler function to our users_list component:

```
<s:DropDownList id="users_list"
    change="usersListChangeHandler(event)"
    dataProvider="{friendsArrayCollection}"
    labelField="name" />
```
Checkins and Facebook Places

Checkin information for a user can be retrieved through their /PROFILE_ID/checkins connection, with a Facebook.api request such as this:

```javascript
Facebook.api(user.id + "/checkins", checkinsResponseHandler);
```

2. When the current selection in users_list component changes, the usersListChangeHandler function is triggered, which requests the selected user’s /checkins connection, like so:

```javascript
function usersListChangeHandler(e:IndexChangeEvent):void {
    var user:FacebookUser = (users_list.selectedItem as FacebookUser);
    Facebook.api(user.id + "/checkins", checkinsResponseHandler);
}
```

3. As with all other Graph API connections, a successful response consists of an Array of Objects; using the fromJSON method we should iterate those Objects into strict-typed FacebookCheckin class instances, using the following code:

```javascript
function checkinsResponseHandler(success:Object, fail:Object):void {
    var results:Array = success as Array;
    if (success && results) {
        for (var i:int = 0; i < results.length; i++) {
            var checkin:FacebookCheckin = FacebookCheckin.
            fromJSON(results[i]);
        }
    }
}
```

By default, the Graph API returns an object containing the properties created_time, place, id, from, and application, similar to the following object:

```json
{
    "created_time": "2011-01-18T13:41:40+0000",
    "place": {
        "location": {
            "city": "Uppingham",
            "zip": "LE15 9NY",
            "street": "1A Uppingham Gate, Ayston Road",
            "longitude": -0.72281,
            "country": "United Kingdom",
            "latitude": 52.58999
        },
        "id": "144796688910339",
```
To place a Marker on our Google Maps component, we first need to extract the Checkin coordinates and use them to create a new LatLng class instance. We can then use that instance to create a new Marker, and then add that marker to our Map component:

```actionscript
var latlng:LatLng = new LatLng(checkin.place.location.latitude, checkin.place.location.longitude);
var marker:Marker = new Marker(latlng);
map.addOverlay(marker);
```

Of course, we can’t simply constantly keep adding new Markers to our map component every time the selected user is changed.

There are many ways to manage markers, but one easy way to do it is to keep track of the active markers by adding them to an Array when they're created, and before adding a new batch of markers, iterate through that Array and remove the old markers.

4. To better manage the Google Map Markers in our application, add a new application-wide variable called `currentMarkers`; a new function, `removeMarkers`; and introduce the highlighted code to the existing `checkinsResponseHandler` function:

```actionscript
var currentMarkers:Array;

function removeMarkers():void {
    if (currentMarkers) {
        for (var i:int = 0; i < currentMarkers.length; i++) {
            map.removeOverlay(currentMarkers[i]);
        }
    }
    currentMarkers = new Array();
}

function checkinsResponseHandler(success:Object, fail:Object):void {
    var results:Array = success as Array;
    if (success && results) {
        removeMarkers();
        for (var i:int = 0; i < results.length; i++) {
```
var checkin:FacebookCheckin = FacebookCheckin.
  fromJSON(results[i]);
  var latlng:LatLng = new LatLng(checkin.place.location.
  latitude, checkin.place.location.longitude);
  var marker:Marker = new Marker(latlng);
  map.addOverlay(marker);
  currentMarkers.push(marker);
}
}

5. We're almost finished—for our final step we want to adjust the position and scale of
the map to focus on those new Markers.

We can do that with the help of the LatLngBounds class, which calculates the required map
center point and zoom level needed to display all of the Markers.

The process for this is to first create an instance of the LatLngBounds class, and then use
the LatLngBounds.extend method to pass in a series of LatLng instances. As we do this,
the dimensions of the LatLngBounds class automatically extend to visually encapsulate
the coordinates we pass in. Once we've passed in all of the Markers, we can use the Map.
setCenter method in conjunction with that LatLngBounds instance to change the map
center and zoom level to appropriate values. To do all of this, we need to make the following
(highlighted) changes to the checkinsResponseHandler function:

function checkinsResponseHandler(success:Object,
  fail:Object):void {
  var results:Array = success as Array;
  if (success && results) {
    removeMarkers();
    var bounds:LatLngBounds = new LatLngBounds();
    for (var i:int = 0; i < results.length; i++) {
      var checkin:FacebookCheckin =
        FacebookCheckin.fromJSON(results[i]);
      var latlng:LatLng = new LatLng(checkin.place.location.
        latitude, checkin.place.location.longitude);
      var marker:Marker = new Marker(latlng);
      map.addOverlay(marker);
      bounds.extend(latlng);
      currentMarkers.push(marker);
    }
    map.setCenter(bounds.getCenter(),
      map.getBoundsZoomLevel(bounds));
  }
}
How it works...

The default visibility of a user's /checkins connection restricts access to those Checkins to only that user's friends, which is why we go through the process of retrieving friends initially—few users go through the process of changing their settings to make their Checkins more publicly available, in my experience.

Plotting Facebook Checkins on a Map is a simple process because the coordinate system of the Checkins matches the coordinates system used by our Google Maps component. The coordinates themselves are not actually the coordinates that the user was physically at when they checked in, but rather the coordinates of the Facebook Place that they checked into, which don't necessarily have to be the same.

What's included in the /PROFILE_ID/checkins connection?
The /PROFILE_ID/checkins connection contains a combination of the Checkins that the user has actually created themselves, and other Checkins that have been made on their behalf (for example, when they are tagged at a location by one of their friends). As far as inclusion in the Graph API connection goes, there is no distinction between being tagged by others and being the instigator of the Check in.

There's more...

Like all other Graph API connections, we can narrow the scope of our request using options such as limit or since, such as when we want to find the most recent location.

Finding the user's most recent Checkin
The Checkins connection naturally returns data with the most recent items being first in the list of responses. So to find the user's most recent Checkin, we can simply add a limit parameter with a value of 1 to our request, like this:

```javascript
var options:Object = new Object();
options.limit = 1;
Facebook.api(users_list.selectedItem.id + "/checkins",
checkinsResponseHandler, options);
```

If we use this limit in conjunction with the rest of the code in this recipe, we'll see that our application now pinpoints the user's last known location, and centers the map around it, selecting the highest-available zoom level for the map at that location.

For More Information:
Exploring Checkins at a specific Place

In addition to exploring a specific user's Checkins, we can also explore Checkins from users at a specific location with the /PAGE_ID/checkins connection, although the same data access restrictions apply to this connection—meaning that unless the current user is a friend of the user who has checked in at that location, we won't see the details of their Checkin.

We can still retrieve a total Checkin count for a Facebook Place through that Place's checkins property, but without being friends with at least one of the users that has checked in to that location, or one of those checked-in users having made all of their Checkins publicly-available by default, the Graph API connection will always return empty results.

Finding Facebook Places near a specific location

In this recipe, we're going to create an application which can search for Facebook Places within a certain distance of specific location coordinates.

Our application will combine the current location of a map component's viewport with the Graph API's Place search capabilities: reading in the current center of the map, constructing a nearby Places search query, parsing the results of that query, and using markers to plot those Places on our map component.

At the end of this recipe, we should have an application which looks a little like this:

For More Information:
Getting ready

The layout of our components for this recipe will be rather simple, consisting of only three components—a List component with an id of place_results; a Button component, load_button; and a Map component, with an id of map.

Requesting Places data from the Graph API requires that our application be authenticated with the Graph API, so to simplify the authentication process, the base class of our application should be the ApplicationBase class, as detailed in the recipe Creating the ApplicationBase class, from Chapter 11, although we don't actually need any specific Extended Permissions from Facebook—just the authentication and the access_token it gives our application.

1. Before we begin searching for nearby Places, we're going to specify the center point of our map component, using following ActionScript function, which will center our map on the location 40.767983, -73.976999 (which corresponds to Central Park, Manhattan):

   ```actionscript
   function onMapReady(e:MapEvent):void {
       var location:LatLng = new LatLng(40.767983, -73.976999);
       map.setCenter(location, 14, MapType.NORMAL_MAP_TYPE);
   }
   ```

2. We can link this function to our MXML component declaration using the following code:

   ```mxml
   <maps:Map id="map" key="[YOUR_API_KEY]"
             mapevent_mapready="onMapReady(event)"
             sensor="true" />
   ```

   The center point of the map doesn't actually matter—it's just going to be easier to specify a location than pick the default map location, which points at the middle of an ocean!

How to do it...

The complete Graph API request URL that we would need to generate in order to search for Places around a geolocation might look like this:

http://graph.facebook.com/search?type=place&center=40.767983,-73.976999

For More Information:

This essentially boils down to a search across the entire Graph API, for objects of type 'place', within the default 1000 m of the center of Central Park.

1. We can construct this request with the following ActionScript code, which uses the value search as the request URL, and the data for the parameters type and center in the request options:

   ```actionscript
   function mapClickHandler(e:Event = null):void {
       var location:LatLng = map.getCenter();
       var options:Object = new Object();
       options.type = "place";
       options.center = location.lat() + "," + location.lng();
       Facebook.api("search", placeSearchResponseHandler, options);
   }
   ```

   As the name of this function—mapClickHandler—implies, this code should be the click handler function for our load_button component. In our application, the user can use the integrated controls to Pan and Zoom their way around the map and, once they’ve focused on a location, click that button to search for nearby places.

2. We can associate the load_button component with this click handler function in MXML, like so:

   ```xml
   <s:Button id="load_button" label="Find nearby Places"
            click="mapClickHandler(event)" />
   ```

3. To parse the responses from this request we would use the following code, which creates an ArrayCollection instance, populated by FacebookPlace class instances:

   ```actionscript
   [Bindable]
   private var placeResults:ArrayCollection;

   function placeSearchResponseHandler(success:Object,
                                         fail:Object):void {
       placeResults = new ArrayCollection();
       var results:Array = success as Array;
       if (success && results) {
           for (var i:int = 0; i < results.length; i++) {
               var place:FacebookPage = FacebookPage.
                               fromJSON(results[i]);
               placeResults.addItem(place);
           }
       }
   }
   ```

For More Information:

To create a marker for a location in the map component, we first define the marker’s coordinates using a LatLng class instance, and then create an instance of type Marker, which is added to the map component, using the following code:

```javascript
var location:LatLng = new LatLng(40.736072, -73.992062);
var marker:Marker = new Marker();
map.addOverlay(marker);
```

4. Add this code to our existing `placeSearchResponseHandler` function and it should look like this:

```javascript
[Bindable]
var placeResults:ArrayCollection;

function placeSearchResponseHandler(success:Object,
fail:Object):void {
    placeResults = new ArrayCollection();
    var results:Array = success as Array;
    if (success && results) {
        for (var i:int = 0; i < results.length; i++) {
            var place:FacebookPage = FacebookPage.
            fromJSON(results[i]);
            var latlng:LatLng = new LatLng(
                place.location.latitude, place.location.longitude);
            var marker:Marker = new Marker(latlng);
            map.addOverlay(marker);
            placeResults.addItem(place);
        }
    }
}
```

5. As before, we can’t keep constantly adding new Markers to our map component every time the user clicks the load_button component. We’ll follow the same marker management practice as in our previous recipe, and add a new variable to the application called `currentMarkers`; a new function, `removeMarkers`; and then make the code changes highlighted in the following code to the existing `placeSearchResponseHandler` function:

```javascript
var currentMarkers:Array;

function removeMarkers():void {
    if (currentMarkers) {
        for (var i:int = 0; i < currentMarkers.length; i++) {
            map.removeOverlay(currentMarkers[i]);
        }
    }
}
```

For More Information:
currentMarkers = new Array();

function placeSearchResponseHandler(success:Object, fail:Object):void {
    placeResults = new ArrayCollection();
    removeMarkers();
    var results:Array = success as Array;
    if (success && results) {
        for (var i:int = 0; i < results.length; i++) {
            var place:FacebookPage = FacebookPage.
fromJSON(results[i]);
            var latlng:LatLng = new LatLng(place.location.
latitude, place.location.longitude);
            var marker:Marker = new Marker(latlng);
            map.addOverlay(marker);
            currentMarkers.push(marker);
            placeResults.addItem(place);
        }
    }
}

This ensures that each time a search is performed, the old markers are removed and a new set of map markers are created, removing any potential for erroneous or duplicated markers.

6. Finally, it would be useful to change the center and zoom level of our Map component to focus better on the Markers we've added. Add the following lines of code to the results parsing function, placeSearchResponseHandler:

    function placeSearchResponseHandler(success:Object, fail:Object):void {
        placeResults = new ArrayCollection();
        removeMarkers();
        var results:Array = success as Array;
        if (success && results) {
            var bounds:LatLngBounds = new LatLngBounds();
            for (var i:int = 0; i < results.length; i++) {
                var place:FacebookPage = FacebookPage.
fromJSON(results[i]);
                var latlng:LatLng = new LatLng(place.location.
latitude, place.location.longitude);
                var marker:Marker = new Marker(latlng);
                map.addOverlay(marker);
                currentMarkers.push(marker);
                placeResults.addItem(place);
                bounds.extend(latlng);
            }
        }
    }

For More Information:
As with our previous recipe, with these additions, we're creating an instance of the LatLngBounds class, looping through the locations and with each successive location we're calling the LatLngBounds.extend method, which extends the dimensions of that LatLngBounds instance, if it's necessary to do so. Once we've iterated through all of the locations returned by our API request, we move and focus the center of the Map component using the setCenter method.

How it works...

A Facebook Place is an odd object to classify—in the Graph API documentation you'll see that there is in fact no standalone 'Place' object, although there are Checkin objects, which then have places listed as a sub-object of the Checkin. Yet within the FQL documentation, there is a dedicated place table available to query for information.

Typically, a Place comes into existence on Facebook as a result of either a location-based Check in, from one of the official Facebook mobile device applications or from a Check in from the Facebook mobile-optimized website. They can also be created by extending a standard user-created Facebook Page, but again only by using the official Facebook.com website.

There's more...

There's one outstanding option that we haven't discussed with our Places search, and that's the distance parameter.

Searching for Places within a specific distance

In addition to the location and type parameters of our search, we can also manually specify a distance from the location, in units of metres, which the results have to be within.

Performing such a search is simply a case of adding a distance property to the request's options Object, such as this:

```javascript
var location:LatLng = map.getCenter();
var options:Object = new Object();
options.type = "place";
```
options.center = location.lat() + "," + location.lng();
options.distance = 100;
Facebook.api("search", placeSearchResponseHandler, options);

If we omit the distance parameter from our request, it defaults to a value of 1 km, giving us a potentially huge number of results (although these will of course be limited by the standard Graph API response length limits).

Creating and using custom Map Markers for our Places

We don't have to use the standard Google Map Markers in our application—instead we can choose to display custom Flash or Flex components, by changing the initial options we supply when we create that marker.

Particularly useful for this recipe, we can use these custom Markers to display a Place's image thumbnail and name, such as this:

A Map Marker can be any class which extends the DisplayObject class. In this case, we're going to combine the familiar workings of the Flex framework's ItemRenderer class and use that as the basis for our Markers.

1. Create a new MXML component, call it PlaceSummaryMarker, and into it add an Image and a Label component, with IDs of place_pic and place_name respectively.

2. Into this component, add the following ActionScript function, which we can use to supply a FacebookPlace class instance, and immediately update the component's image thumbnail and place name:

```actionscript
public function setPlace(value:FacebookPlace):void {
    place_pic.source = Facebook.getImageUrl(value.id);
    place_name.text = value.name;
}
```

3. When we're creating new map marker instances, we can use this component instead of the default, and we can supply it a FacebookPlace instance at the point of its creation, like the following:

```actionscript
var markerIcon:PlaceSummaryMarker = new PlaceSummaryMarker();
markerIcon.setPlace(place);
```

For More Information:
4. To use a custom marker in our map component, we supply a `MarkerOptions` instance as the second parameter of our `Marker` class instance, like this:

```javascript
var markerOptions:MarkerOptions = new MarkerOptions();
var marker:Marker = new Marker(latlng, markerOptions);
```

We supply a reference to this marker through the `MarkerOption` instance's `icon` property, setting the `iconOffset` property to specify the relative center point of our marker, like so:

```javascript
var markerIcon:PlaceSummaryMarker = new PlaceSummaryMarker();

var markerOptions:MarkerOptions = new MarkerOptions();
markerOptions.icon = markerIcon;
markerOptions.iconOffset = new Point(-30, -69);
var marker:Marker = new Marker(latlng, markerOptions);
map.addOverlay(marker);
```

If we supply a `Marker icon` without the `iconOffset` parameter, the point of origin for our marker is the top-left corner of the component. We supply an instance of the `Point` class to act as a relative offset for this top corner—in this case -30 and -69—which tells the Google Maps component to position the component 30 pixels left and 69 pixels upwards of the actual point on the map, aligning the tip of the component’s background graphic with the location on the map.

More complete information about Map Markers, and their options, is best found online in the official documentation for the Google Maps for Flash API:

http://code.google.com/apis/maps/documentation/flash/intro.html

See also

- *Chapter 4, Using the search capabilities of the Graph API*

#### Integrating with HTML5 geolocation capabilities, Maps, and Facebook Places

Modern desktop browsers, like Google Chrome 5+, Firefox 3.5+, Safari 5+, and Internet Explorer 9+ can have scarcely-accurate geolocation capabilities, even without an actual, physical GPS unit attached to your machine. These capabilities are now standardized into HTML5, and with some clever JavaScript and `ExternalInterface` usage our Flash Player application can hook directly in to the browser’s geolocation data capabilities.

For More Information:

Checkins and Facebook Places

In this recipe we're going to build on our earlier recipe, *Finding Facebook Places near a specific location*, and move the center of our map component—and in turn, the center point for our Facebook Places search—based on the location data returned by the user's web browser.

**Getting ready**

The starting point for this application should be the end result of our previous recipe, *Finding Facebook Places near a specific location*. We'll also need to be using a browser that supports the HTML5 geolocation API, such as one of those listed in the previous paragraph.

**How to do it...**

The format of the basic JavaScript code that we would need to execute to request geolocation data from the web browser is standardized in HTML5, and looks like this:

```javascript
navigator.geolocation.getCurrentPosition(success, failure, options);
```

This method asynchronously requests geolocation information from the web browser, supplying a function to call if the request is successful (`success`), a function to call if the request fails (`failure`) and an object of options (`options`) which control the caching and the accuracy requirements of the request.

Further information about the HTML5 Geolocation specification can be found online, at the URL: [http://www.w3.org/TR/geolocation-API/](http://www.w3.org/TR/geolocation-API/)

1. Our first step for this recipe is to create our ActionScript response handler function, which will receive an Object from the JavaScript environment which conforms to the W3C Geolocation Position specification—meaning that the response object it receives will have (at least) two parameters—timestamp and coords.

   We can extract the coordinate values from this response object and use them to create an instance of the Google Map's `LatLng` class with the following code, which simply copies the values directly from the response position `Object` instance and into a new `LatLng` class instance:

   ```javascript
   var location:LatLng = new LatLng(position.coords.latitude,
   position.coords.longitude);
   ```

   Our complete Geolocation response handler function should look like this:

   ```javascript
   function geolocationResponseHandler(position:Object):void {
      var location:LatLng = new LatLng(position.coords.latitude,
      position.coords.longitude);
      map.setCenter(location, map.getMaxZoomLevel(map.
      getCurrentMapType(), location));
   }
   ```
Chapter 10

As with our previous recipe, this `geolocationResponseHandler` function uses our new `LatLng` class instance (location) to adjust the center point and zoom level of the map component on that location.

2. The next step is to make this ActionScript function available to be called by the external JavaScript environment, which we do through use of the `ExternalInterface.addCallback` method, like so:

```ActionScript
ExternalInterface.addCallback("jsResponseHandler",
    geolocationResponseHandler);
```

The first parameter supplied to the `addCallback` method is the name under which our ActionScript function will be available in JavaScript, and the second parameter is a reference to the actual ActionScript function itself.


We should call this code as soon as our application starts up, in our existing `facebookReady` function.

3. With our callback function set up and accessible to JavaScript, we should now request Geolocation information from the browser using the `navigator.geolocation.getCurrentPosition` method and the `ExternalInterface` class.

The `getCurrentPosition` method accepts one, two, or three parameters. We’re actually only going to supply information for the first parameter—the callback function for successful requests—and into that we pass an anonymous JavaScript function which will in turn forward the response object back into our Flash Player application.

The ActionScript code for this request should look like this:

```ActionScript
ExternalInterface.call("navigator.geolocation.getCurrentPosition(function(position){document.getElementById("" + ExternalInterface.objectID + ")\).jsResponseHandler(position)});
```

It looks complex initially, but it’s actually all a single-line ActionScript `String`, which contains a multiline JavaScript method call and nested functions.

For More Information:
Checkins and Facebook Places

The complete code for this recipe should look like this:

```ActionScript
override public function facebookReady():void {
    if (ExternalInterface.available) {
        ExternalInterface.addCallback("jsResponseHandler",
        geolocationResponseHandler);
        ExternalInterface.call( "navigator.geolocation.getCurrentPosition(function(position){document.getElementById(" +
        ExternalInterface.objectID + ").jsResponseHandler(position)})");
    } else {
        trace("ExternalInterface unavailable.");
    }
}

function geolocationResponseHandler(position:Object):void {
    var location:LatLng = new LatLng(position.coords.latitude,
    position.coords.longitude);
    map.setCenter(location, map.getMaxZoomLevel(map.
    getCurrentMapType(), location));
}
```

When we run this application, the user will be asked if they want to allow the current HTML page to access location data, with a message such as this one, from Firefox 4:

![Location Request Message](http://localhost.psyked.co.uk/facebook/FlashPlaces/bin-debug/GeolocationPlaces.html?debug=true)

Once the user has confirmed and allowed access to Geolocation data, the view should re-center itself on the user's current location.

With our location successfully determined, we can now execute the existing `mapClickHandler` function, and retrieve and display information about the nearby Facebook Places.

---

For More Information:

How it works...

The techniques we've used in this recipe essentially tunnel through the external JavaScript interfaces to interact directly with the browser's geolocation support, working with the geolocation APIs specified by the W3C, which are available online at:

http://www.w3.org/TR/geolocation-API/

The resulting object we receive from JavaScript will look like this:

```javascript
{
  "timestamp": 1302348134416,
  "coords": {
    "longitude": -0.7245695,
    "accuracy": 42,
    "altitudeAccuracy": 0,
    "altitude": 0,
    "heading": null,
    "speed": null,
    "latitude": 52.5956211
  }
}
```

The `ExternalInterface` class is normally used for interacting with existing functions in the containing HTML page, but that's not ideal for making our code standalone. Instead we follow a similar approach to that used in the `FacebookJSBridge` ActionScript class, using script-injection techniques to bypass the requirement for specific functions to be present in the containing page.

The `FacebookJSBridge` class is available in the source code of the Facebook ActionScript 3 SDK, under the `com.facebook.graph.core` package, if you're interested to see how this technique is applied on a much larger scale in this library.

For More Information:
Checkins and Facebook Places

There's more...

Of course, what we haven't yet looked at is handling unsuccessful location requests, such as when a user refuses access to their location.

Dealing with Geolocation errors

To handle an error from the Geolocation API, we need to add a second parameter to our initial `navigator.geolocation.getCurrentPosition` request. Following the same response-forwarding technique as the rest of this recipe, our ActionScript code should look like this:

```actionscript
override public function facebookReady():void {
    if (ExternalInterface.available) {
        ExternalInterface.addCallback("jsResponseHandler",
               geolocationResponseHandler);
        ExternalInterface.addCallback("jsResponseError",
               geolocationError);
        ExternalInterface.call( "navigator.geolocation.getCurrentPosition(function(position){document.getElementById(" + ExternalInterface.objectID + ").jsResponseHandler(position)},function(error){document.getElementById(" + ExternalInterface.objectID + ").jsResponseError(error)})");
    } else {
        trace("ExternalInterface unavailable.");
    }
}

function geolocationError(error:Object):void {
    if (error.code == 1) {
        trace("Permission denied");
    } else if (error.code == 2) {
        trace("Geolocation unavailable");
    } else if (error.code == 3) {
        trace("Geolocation timeout");
    }
}
```

If a user refuses to share geolocation information with our application, this `geolocationError` function will be executed, and the response object it returns conforms to the Geolocation API's `PositionError` interface, consisting of an `error` code parameter and an optional `message` parameter.

For More Information:

A sample error response for the user refusing access to Geolocation data looks like this:

```json
{
    "TIMEOUT": 3,
    "POSITION_UNAVAILABLE": 2,
    "PERMISSION_DENIED": 1,
    "code": 1
}
```

**Using an overlay to represent geolocation accuracy**

One of the most common ways to represent the accuracy of geolocation coordinates is to display a circular overlay with a diameter equal to the accuracy, as illustrated in the following image:

![Flash Player Geolocation integration example.](image-url)

For More Information:
We can add these overlays to our Google Maps component with the assistance of the following function, which creates a circular Polygon object:

```javascript
function drawCircle(lat:Number, lng:Number, radius:Number,
    strokeColor:Number = 0xff0000,
    strokeWidth:Number = 2,
    strokeOpacity:Number = 0.5,
    fillColor:Number = 0xff0000,
    fillOpacity:Number = 0.3):void {
    var d2r:Number = Math.PI / 180;
    var r2d:Number = 180 / Math.PI;

    // Convert statute miles into degrees latitude
    var circleLat:Number = radius * 0.014483;
    var circleLng:Number = circleLat / Math.cos(lat * d2r);
    var circleLatLngs:Array = new Array();
    for (var i:Number = 0; i < 33; i++) {
        var theta:Number = Math.PI * (i / 16);
        var vertexLat:Number = lat + (circleLat * Math.sin(theta));
        var vertexLng:Number = lng + (circleLng * Math.cos(theta));
        var latLng:LatLng = new LatLng(vertexLat, vertexLng);
        circleLatLngs.push(latLng);
    }

    var polygonOptions:PolygonOptions = new PolygonOptions();
    var fillStyle:FillStyle = new FillStyle();
    fillStyle.alpha = fillOpacity;
    fillStyle.color = fillColor;
    polygonOptions.fillStyle = fillStyle;

    var strokeStyle:StrokeStyle = new StrokeStyle();
    strokeStyle.alpha = strokeOpacity;
    strokeStyle.color = strokeColor;
    strokeStyle.thickness = strokeWidth;
    polygonOptions.strokeStyle = strokeStyle;

    var polygon:Polygon = new Polygon(circleLatLngs, polygonOptions);
    map.addOverlay(polygon);
}
```

I can't take any credit for this function—it is in fact a slightly modified version of the code found within the Official Google Maps demo application, a live example of which is available at the URL:

http://gmaps-samples-flash.googlecode.com/svn/trunk/demos/CircleDemo/CircleDemo.html

For More Information:

To use this function in our own example, we can add the following code to our 
geolocationResponseHandler handler function:

```actionscript
function geolocationResponseHandler(position:Object):void {
    var location:LatLng = new LatLng(position.coords.latitude,
    position.coords.longitude);
    map.setCenter(location, map.getMaxZoomLevel(map.
    getCurrentMapType(), location));
    map.addOverlay(new Marker(location));
    drawCircle(location.latitude, location.longitude, (position.
    coords.accuracy / 2) / 1609.344);
}
```

This additional code passes in the latitude and longitude parameters for the location, 
along with the desired radius of the circle overlay. To obtain the overlay's radius, we convert 
our Geolocation accuracy value (which is measured in meters) into a value in units of miles 
(which is what the drawCircle function expects) by first dividing the accuracy value in half 
(diameter → radius) and then by 1,609.344 (meters → miles).

**Working with the native Flash Platform Geolocation capabilities**

Native Flash Platform support for device sensors such as GPS or Geolocation data is available 
in the AIR 2+ runtime for mobile devices. It is not, unfortunately, supported for AIR for Desktop 
or basic in-browser Flash Platform applications (yet).

The ActionScript 3 Geolocation class comes as part of the AIR runtime libraries. To check 
whether Geolocation information is available for the current device we can use using the 
Geolocation.isSupported method.

If data is available, we create a new instance of the Geolocation class, set an update 
interval for that instance, and register listeners to look for those update Events. The code 
that we'd need to use to do this should look like this:

```actionscript
if (Geolocation.isSupported) {
    var location:Geolocation = new Geolocation();
    location.setRequestedUpdateInterval(100);
    location.addEventListener(GeolocationEvent.UPDATE,
    locationUpdateHandler);
}
```

The event that comes through to this handler function is of type GeoLocationEvent, 
containing the latest latitude and longitude information. To take this data and use it 
to change the focal point of our map component, we could use the following function:

```actionscript
function geolocationUpdateHandler(e:GeolocationEvent):void {
    var location:LatLng = new LatLng(e.latitude, e.longitude);
    map.setCenter(location, map.getMaxZoomLevel( map.
    getCurrentMapType(), location));
}
```

For More Information:  
Checkins and Facebook Places

The GeolocationEvent returns an object with properties for altitude, heading, horizontalAccuracy, latitude, longitude, speed, timestamp, and verticalAccuracy—depending slightly on the capabilities of the device, of course.

Geolocation updates are asynchronous—meaning that we won't get an immediate location update when we instantiate the Geolocation class. Instead we specify a desired frequency of updates, using the setRequestedUpdateInterval method, and the locationUpdateHandler function that we associated with the update event will receive this information as-and-when the device passes that information into our application.

The update frequency for these events depends on the device itself. Some devices, like the first generation iPhones, don’t have proper GPS capabilities, but instead identify their location via cell-towers and suchlike. As a result the accuracy of their locations is much lower, and the frequency with which they update is also much lower.


See also

- Finding Facebook Places near a specific location

Creating new Checkins at an existing Facebook Place

We may not be able to create new Facebook Places through the Graph API, but with the correct Extended Permissions for the current user, we can create new Checkins at existing Facebook Places.

In this recipe, we're going to build an interface which allows us to browse for, and check into, an existing location on the user's behalf.

The first step for our application would be to retrieve a list of the eligible locations for the current user to check in at. Our interface itself will allow the user to browse around a Map containing Markers representing the nearby Facebook Places. Clicking on a Marker will select it, and once a Marker is selected, the user can click a Check in here button to check in at that location.

For More Information:
Our finished application should look something like this:

![Map of nearby places](image.png)

**Getting ready**

To create a Checkin on the user's behalf, our application will require the `publish_checkins` Extended Permission from the logged in user. To simplify the authentication process, the base class of our application should be the `ApplicationBase` class, detailed in the recipe *Creating the ApplicationBase class* from Chapter 11.

In our application interface, we will need three main components: two `Button` components, with `ids` of `nearby_places_btn` and `checkin_here_btn`, which will be used to initiate a nearby Places search and create a new Checkin respectively, along with a Google Maps component (called `map`) to display the Places themselves.

1. Building on our earlier recipes, we can use the following ActionScript function to retrieve Place locations near to the current center of our map component:

   ```actionscript
   function findNearbyPlaces(e:Event = null):void {
       var location:LatLng = map.getCenter();
       var options:Object = new Object();
       options.type = "place";
   }
   ```

For More Information:

options.center = location.lat() + "," + location.lng();
Facebook.api("search", nearbyPlacesResponseHandler, options);
}

2. We can link this function in as the click handler of the nearby_places_btn using the following MXML:

```xml
<s:Button id="nearby_places_btn"
    label="Find nearby Places"
    click="findNearbyPlaces(event)" />
```

3. Once we have retrieved these Place details, we can plot them on our Map component with the following code:

```actionscript
var currentMarkers:Array;

function nearbyPlacesResponseHandler(success:Object,
    fail:Object):void {
    var results:Array = success as Array;
    if (success && results) {
        removeMarkers();
        for (var i:int = 0; i < results.length; i++) {
            var place:FacebookPage = FacebookPage.
                fromJSON(results[i]);
            var latlng:LatLng = new LatLng(place.location.
                latitude, place.location.longitude);
            var markerOptions:MarkerOptions = new MarkerOptions();
            var markerComponent:PlaceSummaryMarker = new
                PlaceSummaryMarker();
            markerComponent.data = place;
            markerOptions.icon = markerComponent;
            markerOptions.iconOffset = new Point(-30, -69);
            var marker:Marker = new Marker(latlng, markerOptions);
            map.addOverlay(marker);
            currentMarkers.push(marker);
        }
    }
}

function removeMarkers():void {
    if (currentMarkers) {
        for (var i:int = 0; i < currentMarkers.length; i++) {
            map.removeOverlay(currentMarkers[i]);
        }
    }
    currentMarkers = new Array();
}
```

For More Information:

This ActionScript code also uses a custom marker component—an instance of the `PlaceSummaryMarker` class—that we created as part of our previous recipe, *Finding Facebook Places near a specific location*.

### How to do it...

With our setup so far, we've already got an interface which can retrieve nearby Places and plot them on a map, so the next step for this recipe is to make it possible to visually select one of those Places to check into.

We're going to make the map markers, each of which is an instance of our custom map marker class (`PlaceSummaryMarker`), selectable by the user through mouse click events. Only one Marker should be selectable at a single time, as only one Place can be checked into at any time.

1. To make our map markers modally-selectable, we would need to add the following ActionScript code to our application, which manages the visual selection styles and modal-selection behavior of all of the map markers:

   ```actionscript
   var selectedMarker:PlaceSummaryMarker;
   var currentMarkers:Array;

   function nearbyPlacesResponseHandler(success:Object, fail:Object):void {
     var results:Array = success as Array;
     if (success && results) {
       removeMarkers();
       for (var i:int = 0; i < results.length; i++) {
         var place:FacebookPlace = FacebookPlace.fromJson(results[i]);
         var latlng:LatLng = new LatLng(place.location.latitude, place.location.longitude);
         var markerOptions:MarkerOptions = new MarkerOptions();
         var markerComponent:PlaceSummaryMarker = new PlaceSummaryMarker();
         markerComponent.setPlace(place);
         markerOptions.icon = markerComponent;
         markerOptions.iconOffset = new Point(-30, -69);
         var marker:Marker = new Marker(latlng, markerOptions);
         marker.addEventListener(MapMouseEvent.CLICK, markerClickHandler);
         map.addOverlay(marker);
         currentMarkers.push(marker);
       }
     }
   }
   
   For More Information:
function markerClickHandler(e:MapMouseEvent):void {
  deselectMarkers();
  var marker:Marker = e.currentTarget as Marker;
  var icon:PlaceSummaryMarker = marker.getOptions().icon as PlaceSummaryMarker;
  map.panTo(e.latLng);
  selectedMarker = icon;
  selectedMarker.currentState = "selected";
}

function removeMarkers():void {
  if (currentMarkers) {
    for (var i:int = 0; i < currentMarkers.length; i++) {
      map.removeOverlay(currentMarkers[i]);
      currentMarkers[i].removeEventListener(MapMouseEvent.CLICK, markerClickHandler);
    }
  }
  currentMarkers = new Array();
  deselectMarkers();
}

function deselectMarkers():void {
  if (selectedMarker) {
    selectedMarker.currentState = "normal";
  }
}

2. Our custom marker class—PlaceSummaryMarker—already keeps track of the FacebookPlace instance it represents, so our next task is to add a click event
handler to each of these Markers, and then in that event handler function extract
that value of into the currentPlace parameter:

var currentPlace:FacebookPlace;

function markerClickHandler(e:MapMouseEvent):void {
  deselectMarkers();
  var marker:Marker = e.currentTarget as Marker;
  var icon:PlaceSummaryMarker = marker.getOptions().icon as PlaceSummaryMarker;
  map.panTo(e.latLng);
  currentPlace = icon.data;
  selectedMarker = icon;
}
With a reference to a specific FacebookPlace instance stored in this currentPlace variable, we now have enough information to create a Checkin on the user's behalf. A request to create a new Checkin requires two parameters—the Facebook Place id and coordinates of the current user (or the Place itself).

3. To create a Checkin on the user's behalf, we can create a new function, createCheckin, and it should look like this:

```actionscript
function createCheckin(e:Event = null):void {
    var coords:Object = new Object();
    coords.latitude = currentPlace.location.latitude;
    coords.longitude = currentPlace.location.longitude;

    var options:Object = new Object();
    options.place = currentPlace.id;
    options.coordinates = JSON.encode(coords);
    Facebook.api("me/checkins", checkinResponseHandler, options, "post");
}
```

4. Finally, we should attach this function to our Button component with following MXML property:

```xml
<s:Button id="checkin_here_btn" label="Check in here"
    click="createCheckin(event)" />
```

With this request, we've now created a Checkin on the user's behalf at that Place. Add a final response handler for that request and our application should be complete.

**How it works...**

We create a new Checkin by issuing a POST request to the user's own /checkins connection, supplying, at the very least, the id of an existing Facebook Place under the parameter place, and the user's current location, under the parameter coordinates.

Beyond this, there are no checks as to the validity of the Checkin, so it's quite possible that we could intentionally spoof Checkin locations, such as creating Checkins to places like 'Area 51' (id: 14742924194882).

We can create a Checkin on a user's behalf at any existing Facebook Place but, unfortunately, we can't create our own Place from scratch—at least, not yet; it remains to be seen when and whether this capability will be opened up to developers working with the Graph API.

For More Information:

There's more...

In addition to simply checking the current user in at a location, the Graph API also supports two additional properties: message and tags.

Adding a message to a Checkin

We can add a message to each Checkin simply by supplying a message parameter when we post a new Checkin to the Graph API, like this:

```javascript
var coords:Object = new Object();
coords.latitude = currentPlace.location.latitude;
coords.longitude = currentPlace.location.longitude;

var options:Object = new Object();
options.place = currentPlace.id;
options.coordinates = JSON.encode(coords);
options.message = "My awesome checkin status.";
Facebook.api("me/checkins", checkinReponseHandler, options, "post");
```

This message is simple plain-text String, much like we use when we create a status update for the user—meaning that there's no support for complex text formatting or file upload support (such as a Photo) with a Checkin.

"Tagging" other users to include them in a Checkin

As well as creating a Checkin for the current user, we can also create a Checkin on our friend's behalf, by adding an Array of their user IDs to the request, under the tags parameter of our new Checkin, like this:

```javascript
var coords:Object = new Object();
coords.latitude = currentPlace.location.latitude;
coords.longitude = currentPlace.location.longitude;

var options:Object = new Object();
options.place = currentPlace.id;
options.coordinates = JSON.encode(coords);
options.tags = new Array("10000192832487", "10000192832488");
Facebook.api("me/checkins", checkinReponseHandler, options, "post");
```

With this request, the user IDs in the tags parameter will also be checked into the location you've specified, under the same Facebook Checkin object.

See also

- Finding Facebook Places near a specific location
- Chapter 4: Creating, Editing and Deleting Graph API objects

For More Information:

Where to buy this book

You can buy Flash Facebook Cookbook from the Packt Publishing website:

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.