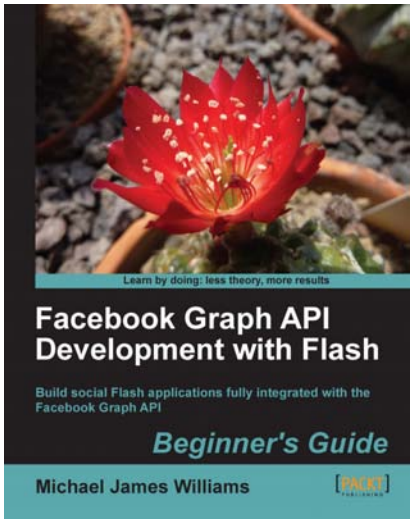
[PACKT] PUBLISHING

Facebook Graph API Development with Flash Beginner's Guide

Michael James Williams



Chapter No. 2
"Welcome to the Graph"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.2 "Welcome to the Graph"

A synopsis of the book's content

Information on where to buy this book

About the Author

Michael James Williams is a technical concept writer and freelance Flash developer. He is the technical editor for the tutorial website Activetuts+, and also runs his own blog about Flash game development.

He currently lives in England, in a nice litt le town that has both a river and a canal, and has been using Facebook since it was just some site that his American housemate wouldn't stop talking about.

You can follow Michael on Twitt er at `http://twitter.com/MichaelJW`.

His public Facebook profile is available at `http://on.fb.me/MichaelJamesWilliams`.

Activetuts+ can be found at `http://active.tutsplus.com/`.

Michael's website is `http://michaeljameswilliams.com/`.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Facebook Graph API Development with Flash Beginner's Guide

Facebook is big, by all meanings of the word. It's used by half a billion people—and countless businesses, bands, and public figures—for socializing and self-promotion. It's also a huge development platform, with tens of thousands of applications.

It's now common to see a Facebook "Like" button on blog posts, news articles, and many other websites. In the same way, Facebook integration is becoming more and more desirable for browser-based RIAs and games, with some, like FarmVille, even being based entirely around Facebook. That's where Flash comes in.

What This Book Covers

Chapter 1, *Introduction*, gets you up to speed with Facebook and ready to learn to develop Flash applications that connect with the Facebook platform. You'll learn why it's worth putting more time into developing for Facebook than other social networks (and why it's likely to stay that way), and get yourself technically set up for coding.

Chapter 2, *Welcome to the Graph*, introduces you to Facebook's model for connecting all the information in its huge data stores—the **Graph API**. You'll discover how intuitive this model is, and will start to explore the publicly available data using AS3 through utility code, which you'll build from scratch.

Chapter 3, *Let Me In!*, breaks down Facebook's systems for security, permissions, and authentication. You'll learn how to access the private data of Facebook users (including their photos, biographical information, and lists of friends). You will also start using the official Adobe ActionScript 3 SDK for Facebook platform alongside your own utility code.

Chapter 4, *Digging Deeper into the Graph*, helps you understand the concepts of paging and filtering, so that you aren't restricted to using only the default dataset that Facebook presents you with. You'll find out how to obtain data from specified dates, and how to speed up your applications by retrieving information from multiple sources at once.

Chapter 5, *Search Me*, builds on the previous chapter by teaching you how to search for data based on criteria other than dates. You'll learn how to retrieve Wall Posts by specific users, pages with specific names, and places by specific geographical coordinates.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Chapter 6, *Adding to the Graph*, takes you beyond merely retrieving data and into publishing new data to Facebook. You'll find out how to create new Wall Posts (including rich posts including images and embedded hyperlinks); how to comment on other users' Wall Posts; how to create new events, notes, and albums; and how to upload photos from your hard drive.

Chapter 7, *FQL Matters*, takes a break from the Graph API to teach you how to learn a powerful search tool—**Facebook Query Language**. You'll trade the Graph API's intuitiveness and simplicity for FQL's depth and additional features, while also understanding the benefits that each approach offers over the other.

Chapter 8, *Finishing Off*, wraps up what you've learned throughout the book and gets you ready to release your application to the wild. You'll find out how to embed your application into the Facebook website itself; how to get it into the official Facebook Application Directory; and how to export it as a desktop or Android application, while still keeping its Facebook connectivity. Finally, you'll learn how to keep up-to-date with the ever-changing Facebook platform, and discover some useful resources for taking what you've learned even further.

Appendix, *Pop Quiz Answers*, contains answers to all the Pop Quizzes in the book

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

2
Welcome to the Graph

Facebook has a huge store of information, on people, companies, events, photo albums, and more. It also knows how all of these are linked: which person owns each album; which people appear in each photo; which company is organizing each event.

For four years, this was accessed using a huge, sprawling API, which got more complex as new abilities were bolted on to it. In April 2010, Facebook launched the **Graph API**, greatly simplifying how developers can retrieve all of this data.

In this chapter we shall:

- Explore the Facebook Graph
- Learn what the Graph API is, and how it structures all the data on Facebook
- Access public Graph data using AS3 and the Graph API

So let's get on with it.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Accessing the Graph API through a Browser

We'll dive right in by taking a look at how the Graph API represents the information from a public **Page**.



When I talk about a Page with a capital P, I don't just mean any web page within the Facebook site; I'm referring to a specific type of page, also known as a public profile. Every Facebook user has their own personal profile; you can see yours by logging in to Facebook and clicking on the "Profile" link in the navigation bar at the top of the site. Public profiles look similar, but are designed to be used by businesses, bands, products, organizations, and public figures, as a way of having a presence on Facebook.

This means that many people have both a personal profile and a public profile. For example, Mark Zuckerberg, the CEO of Facebook, has a personal profile at `http://www.facebook.com/zuck` and a public profile (a Page) at `http://www.facebook.com/markzuckerberg`. This way, he can use his personal profile to keep in touch with his friends and family, while using his public profile to connect with his fans and supporters.

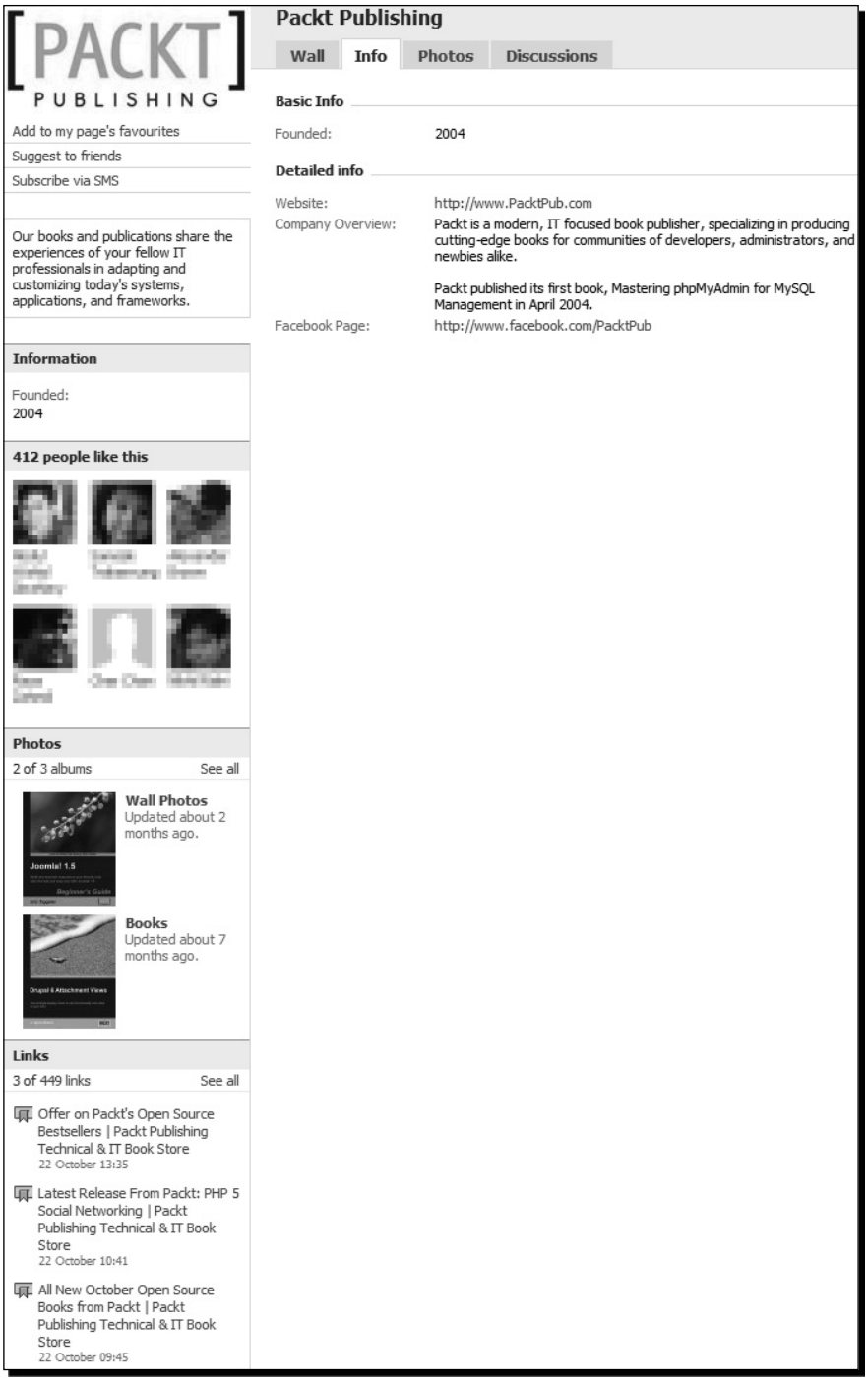
There is a second type of Page: a **Community Page**. Again, these look very similar to personal profiles; the difference is that these are based on topics, experience, and causes, rather than entities. Also, they automatically retrieve information about the topic from Wikipedia, where relevant, and contain a live feed of wall posts talking about the topic.

All this can feel a little confusing – don't worry about it! Once you start using it, it all makes sense.

Time for action – loading a Page

Browse to `http://www.facebook.com/PacktPub` to load Packt Publishing's Facebook Page. You'll see a list of recent wall posts, an **Info** tab, some photo albums (mostly containing book covers), a profile picture, and a list of fans and links.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book



For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

That's how website users view the information. How will our code "see" it? Take a look at how the Graph API represents Packt Publishing's Page by pointing your web browser at `https://graph.facebook.com/PacktPub`. This is called a **Graph URL** – note that it's the same URL as the Page itself, but with a secure `https` connection, and using the `graph` sub domain, rather than `www`.

What you'll see is as follows:

```
{
   "id": "204603129458",
   "name": "Packt Publishing",
   "picture": "http://profile.ak.fbcdn.net/hprofile-ak-snc4/
     hs302.ash1/23274_204603129458_7460_s.jpg",
   "link": "http://www.facebook.com/PacktPub",
   "category": "Products_other",
   "username": "PacktPub",
   "company_overview": "Packt is a modern, IT focused book publisher,
     specializing in producing cutting-edge books for communities of
     developers, administrators, and newbies alike.\n\nPackt
     published its first book, Mastering phpMyAdmin for MySQL
     Management in April 2004.",
   "fan_count": 412
}
```

What just happened?

You just fetched the Graph API's representation of the Packt Publishing Page in your browser.

The Graph API is designed to be easy to pick up – practically self-documenting – and you can see that it's a success in that respect. It's pretty clear that the previous data is a list of fields and their values.

The one field that's perhaps not clear is `id`; this number is what Facebook uses internally to refer to the Page. This means Pages can have two IDs: the numeric one assigned automatically by Facebook, and an alphanumeric one chosen by the Page's owner. The two IDs are equivalent: if you browse to `https://graph.facebook.com/204603129458`, you'll see exactly the same data as if you browse to `https://graph.facebook.com/PacktPub`.

Have a go hero – exploring other objects

Of course, the Packt Publishing Page is not the only Page you can explore with the Graph API in your browser. Find some other Pages through the Facebook website in your browser, then, using the `https://graph.facebook.com/id` format, take a look at their Graph API representations. Do they have more information, or less?

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Next, move on to other types of Facebook objects: personal profiles, events, groups. For personal profiles, the `id` may be alphanumeric (if the person has signed up for a custom Facebook Username at `http://www.facebook.com/username/`), but in general the `id` will be numeric, and auto-assigned by Facebook when the user signed up.

For certain types of objects (like photo albums), the value of `id` will not be obvious from the URL within the Facebook website; we'll look at how to find these later in the chapter.

In some cases, you'll get an error message, like:

```
{
   "error": {
      "type": "OAuthAccessTokenException",
      "message": "An access token is required to request
        this resource."
   }
}
```

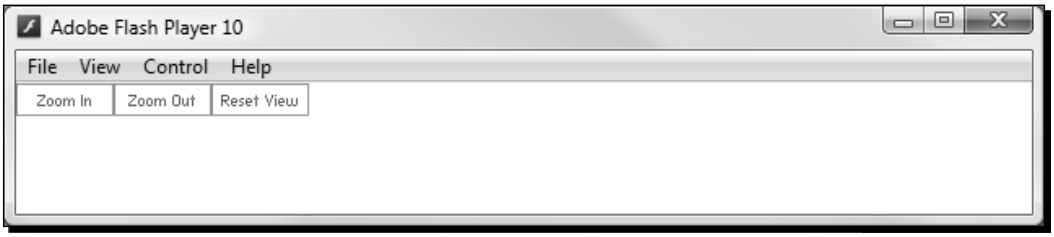
Again, we'll look at what this means and how to get around it later in the book.

Accessing the Graph API through AS3

Now that you've got an idea of how easy it is to access and read Facebook data in a browser, we'll see how to fetch it in AS3.

Time for action – retrieving a Page's information in AS3

Set up the project from the Chapter 2 start files, as explained in Chapter 1. Check that the project compiles with no errors (there may be a few warnings, depending on your IDE). You should see a 640 x 480 px SWF, all white, with just three buttons in the top-left corner: **Zoom In**, **Zoom Out**, and **Reset View**:



This project is the basis for a Rich Internet Application (RIA) that will be able to explore all of the information on Facebook using the Graph API. All the code for the UI is in place, just waiting for some Graph data to render. Our job is to write code to retrieve the data and pass it on to the renderers.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

I'm not going to break down the entire project and explain what every class does, as the focus of this book is on using Facebook with Flash, not on building RIAs. What you need to know at the moment is a single instance of the `controllers.CustomGraphContainerController` class is created when the project is initialized, and it is responsible for directing the flow of data to and from Facebook. It inherits some useful methods for this purpose from the `controllers.GCController` class; we'll make use of these later on.

Open the `CustomGraphContainerController` class in your IDE. It can be found in `\src\controllers\CustomGraphContainerController.as`, and should look like the listing below:

```
package controllers
{
  import ui.GraphControlContainer;

  public class CustomGraphContainerController extends GCController
  {

    public function CustomGraphContainerController
      (a_graphControlContainer:GraphControlContainer)
    {
      super(a_graphControlContainer);
    }

  }

}
```

The first thing we'll do is grab the Graph API's representation of Packt Publishing's Page via a Graph URL, like we did using the web browser. For this we can use a `URLLoader`.



The `URLLoader` and `URLRequest` classes are used together to download data from a URL. The data can be text, binary data, or URL-encoded variables.

The download is triggered by passing a `URLRequest` object, whose `url` property contains the requested URL, to the `load()` method of a `URLLoader`.

Once the required data has finished downloading, the `URLLoader` dispatches a `COMPLETE` event. The data can then be retrieved from its `data` property.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Modify `CustomGraphContainerController.as` like so (the highlighted lines are new):

```
package controllers
{
  import flash.events.Event;
  import flash.net.URLLoader;
  import flash.net.URLRequest;
  import ui.GraphControlContainer;

  public class CustomGraphContainerController extends GCController
  {

    public function CustomGraphContainerController
     (a_graphControlContainer:GraphControlContainer)
    {
      super(a_graphControlContainer);

      var loader:URLLoader = new URLLoader();
      var request:URLRequest = new URLRequest();
      //Specify which Graph URL to load
      request.url = "https://graph.facebook.com/PacktPub";
      loader.addEventListener(Event.COMPLETE,
        onGraphDataLoadComplete);
      //Start the actual loading process
      loader.load(request);
    }

    private function onGraphDataLoadComplete(a_event:Event):void
    {
      var loader:URLLoader = a_event.target as URLLoader;
      //obtain whatever data was loaded, and trace it
      var graphData:String = loader.data;
      trace(graphData);
    }

  }

}
```

All we're doing here is downloading whatever information is at `https://graph.facebook.com/PackPub` and tracing it to the output window.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Test your project, and take a look at your output window. You should see the following data:

```
{"id":"204603129458","name":"Packt Publishing","picture":"http:\/\/
profile.ak.fbcdn.net\/hprofile-ak-snc4\/hs302.
ash1\/23274_204603129458_7460_s.jpg","link":"http:\/\/www.facebook.
com\/PacktPub","category":"Products_other","username":"PacktPub",
"company_overview":"Packt is a modern, IT focused book publisher,
specializing in producing cutting-edge books for communities of
developers, administrators, and newbies alike.\n\nPackt published
its first book, Mastering phpMyAdmin for MySQL Management in April
2004.","fan_count":412}
```

If you get an error, check that your code matches the previously mentioned code. If you see nothing in your output window, make sure that you are connected to the Internet. If you still don't see anything, it's possible that your security settings prevent you from accessing the Internet via Flash, so check those.

What just happened?

The line breaks and tabulation between values have been lost, and some characters have been escaped, making it hard to read… but you can see that this is the same data as we obtained when browsing to `https://graph.facebook.com/PacktPub`. No surprise here; that's all the `URLLoader` does.

The data's not very useful to us in that form. In order to do something with it, we need to convert it to an object that we can interact with natively in AS3.

The format which Graph API uses is called **JSON** (**JavaScript Object Notation**; pronounced "Jason").

JSON is a human-readable, text-based data format standard. It allows you to represent objects as key-value pairs, like so:

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

The values can be strings (enclosed in quote marks), or numbers, Boolean values, or `null` (not enclosed in quote marks).

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

JSON objects can also contain arrays, using square brackets:

```
{
  "key1": "value1",
  "array":
  [
    "First item in array",
    "Second item in array",
    "Third item in array"
  ]
}
```

They can even contain other JSON objects, by nesting curly braces:

```
{
  "key1": "value1",
  "subObject":
  {
    "subKey1": "subValue1",
    "subKey2": "subValue2",
  }
}
```

These sub-objects can contain other objects and arrays, and arrays can contain other objects or arrays, too.

Note that this is very similar to the AS3 syntax for declaring an object:

```
var as3Object:Object = {
  key1:"value1",
  key2:"value2",
  subObject:{
    subKey1:"subValue1"
  },
  myArray:[1, 2, 3]
}
```



For more information, check out `http://www.json.org`.

Unlike with XML, AS3 has no native features for handling JSON objects – but there is an officially supported library that does.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Time for action – deserializing a JSON object

Adobe's **as3corelib** library contains a set of utility classes for serializing and deserializing JSON. It's available at `http://github.com/mikechambers/as3corelib`, but you don't need to download it, as it is already included in the `\src\` directory of the project. (It consists of every class in `com.adobe.*`)

1. In `CustomGraphContainerController.as`, import the `JSON` class:

```
import com.adobe.serialization.json.JSON;
```

2. Modify the `onGraphDataLoadComplete()` function so that it deserializes the JSON string to an object, instead of simply tracing the string:

```
private function onGraphDataLoadComplete(a_event:Event):void
{
  var loader:URLLoader = a_event.target as URLLoader;
  //obtain whatever data was loaded, and trace it
  var graphData:String = loader.data;
  var decodedJSON:Object = JSON.decode(graphData);
}
```

3. Trace the `name` property of this new object, to check that it worked:

```
private function onGraphDataLoadComplete(a_event:Event):void
{
  var loader:URLLoader = a_event.target as URLLoader;
  //obtain whatever data was loaded, and trace it
  var graphData:String = loader.data;
  var deserialisedJSON:Object = JSON.decode(graphData);
  trace("name:", decodedJSON.name);
}
```

4. Compile and run the SWF. Resulting output:

```
name: Packt Publishing
```

What just happened?

We passed this string to the `JSON.decode()` method:

```
{
   "id": "204603129458",
   "name": "Packt Publishing",
   "picture": "http://profile.ak.fbcdn.net/hprofile-ak-snc4/
     hs302.ash1/23274_204603129458_7460_s.jpg",
   "link": "http://www.facebook.com/PacktPub",
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
    "category": "Products_other",
    "username": "PacktPub",
    "company_overview": "Packt is a modern, IT focused book
      publisher, specializing in producing cutting-edge books for
      communities of developers, administrators, and newbies
      alike.\n\nPackt published its first book, Mastering
      phpMyAdmin for MySQL Management in April 2004.",
    "fan_count": 412
}
```

and it turned the string into a native AS3 object, as if we had typed this:

```
var graphObject:Object = {};
graphObject.id = "204603129458";
graphObject.name = "Packt Publishing";
graphObject.picture = "http://profile.ak.fbcdn.net/hprofile-ak-
snc4/hs302.ash1/23274_204603129458_7460_s.jpg";
graphObject.link = "http://www.facebook.com/PacktPub";
graphObject.category = "Products_other";
graphObject.username = "PacktPub";
graphObject.company_overview = "Packt is a modern, IT focused
book publisher, specializing in producing cutting-edge books for
communities of developers, administrators, and newbies alike.\n\
nPackt published its first book, Mastering phpMyAdmin for MySQL
Management in April 2004."
graphObject.fan_count = 412;
```

(Note that unlike the raw string we had earlier, the slashes in the URLs have not been escaped.)

This means we can easily access any of the information Facebook has about this Page, or even iterate through every piece of data.

Time for action – visualizing the info

Enough traces! It's time we displayed something in our actual SWF.

`CustomGraphContainerController` inherits a method called `renderGraphObject()` which will take care of this for us. All we have to do is pass it an argument of type `graph.GraphObject`.

`GraphObject.as` is a simple class; feel free to open it and take a look:

```
package graph
{
  import graph.controls.GraphObjectRenderer;
  public dynamic class GraphObject extends BaseGraphItem
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
    {
      public var rendererObject:GraphObjectRenderer;
      public var graphObjectListRenderers:Array = [];

      public function GraphObject()
      {

      }

    }

  }
```

Honestly, there's no need to worry about any of the code there. All you need to know is that it's `dynamic`, which means that we can assign new properties to it during runtime, without having to specify their names beforehand. So we can do this:

```
var graphObject:GraphObject = new GraphObject();
graphObject.favoriteColor = "red";
```

When a `GraphObject` is passed to the `CustomGraphContainerController.renderGraphObject()` method, every single property of the `GraphObject` will be rendered in a fancy list, automatically. Every single property apart from the two that are defined in the class already, that is!

So what we have to do, inside `CustomGraphContainerController.onGraphDataLoadComplete()`, is:

1. Create a new instance of `GraphObject`.
2. Copy all the properties of `decodedJSON` to this new `GraphObject`.
3. Pass the `GraphObject` to `renderGraphObject()`.
4. The code for doing that is as follows:

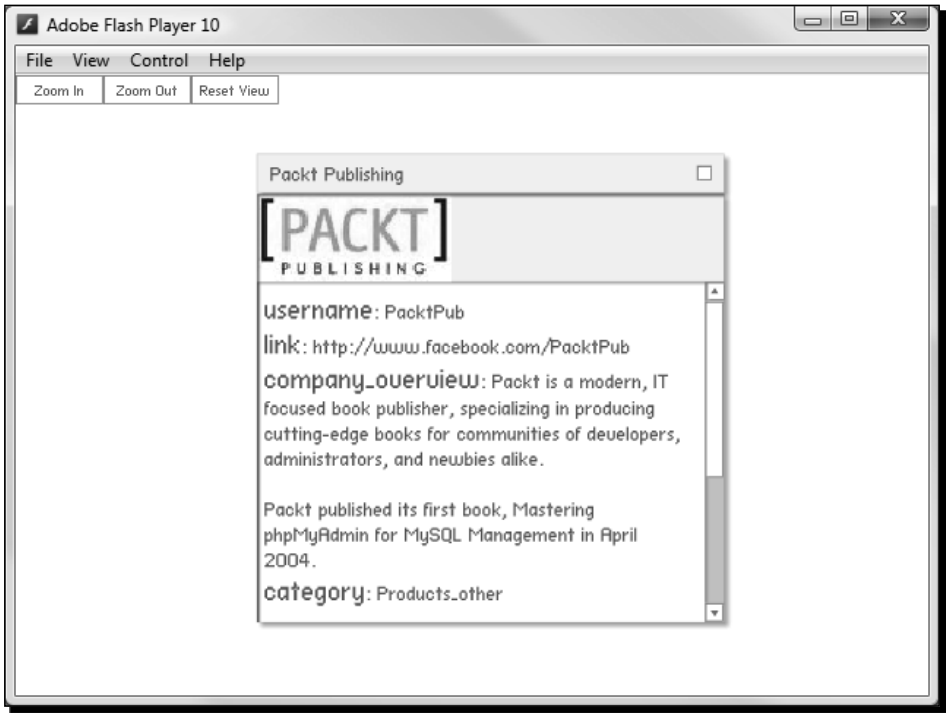
```
private function onGraphDataLoadComplete(a_event:Event):void
{
  var loader:URLLoader = a_event.target as URLLoader;
  //obtain whatever data was loaded, and trace it
  var graphData:String = loader.data;
  var decodedJSON:Object = JSON.decode(graphData);

  var graphObject:GraphObject = new GraphObject();
  //copy all the properties from decodedJSON to graphObject
  for (var key:String in decodedJSON)
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
    {
      graphObject[key] = decodedJSON[key];
    }
    this.renderGraphObject(graphObject);
  }
```

5. Compile and test. The SWF is shown in the next screenshot:



You can click the **Zoom In** button a few times to make the Renderer larger and clearer, as in the screenshot above. Your Renderer might display the fields in a different order than depicted; Facebook returns the fields in an arbitrary order.

What just happened?

The window that appeared on stage is what I call a Renderer – specifically, a **Graph Object Renderer**. It can be dragged around by the title bar, the contents can be scrolled, and you can close it by clicking the button in the top-right corner.

So, you've successfully fetched data from Facebook's Graph API and displayed it in a SWF. Your SWF is flexible; change `request.url` to point to the Graph URL of a different Facebook object and you'll see it displayed in the **Renderer**.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Most of the data from the `GraphObject` is displayed in a text area inside the window, in a simple "`key: value`" format. The Page's `name` field is displayed in the window's title bar, and if the Page has a `picture` field (we can see from the JSON that `PacktPub` does), the image is downloaded and displayed inside the renderer using a `Loader`.



Like `URLLoader`, the `flash.display.Loader` class downloads the object that a given `URLRequest` points to, dispatching a `COMPLETE` event when ready. Unlike `URLLoader`, Loader is used to download images and SWFs, and the event is actually dispatched by one of its sub-objects, `contentLoaderInfo`. Also, `Loader` extends `DisplayObject`, and takes the appearance of the image when it has finished downloading.

Flash's security model prevents an image's data being accessed by SWFs residing on a different domain than the image itself, unless there is a cross-domain policy file on the domain of the image that allows it. Fortunately, Facebook's cross-domain policy file is lenient, allowing such access by every domain.

So, really, this is just a graphical way of representing a Page object from the Graph API.

Understanding connections

"That's all well and good," you may be thinking, "but it doesn't show all the data associated with the Page, does it? Where are the wall posts and photos?"

Time for action – finding connections in a browser

Facebook treats wall posts, photos, videos, and even statuses as separate objects within the Graph API, rather than jamming them all into a single Page object. For instance, here's an object representing a single Post by Packt Publishing:

```
{
   "id": "204603129458_127056137323572",
   "from": {
      "name": "Packt Publishing",
      "category": "Products_other",
      "id": "204603129458"
   },
   "message": "The Amazon SimpleDB Developer Guide has been published!
     Get your copy now! http://bit.ly/blFQUG",
   "picture": "http://external.ak.fbcdn.net/
     safe_image.php?d=c4a7887cb52dd8f93e439aaec13c034b&w=130&h=130&url
     =https%3A%2F%2Fwww.packtpub.com%2Fsites%2Fdefault%2Ffiles%2Fimage
     cache%2Fproductview%2F7344EN_MockupCover%2520Template.jpg",
   "link": "http://bit.ly/blFQUG",
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
    "name": "Amazon SimpleDB Developer Guide | Packt Publishing
        Technical & IT Book Store",
    "caption": "bit.ly",
    "description": "Gain in-depth understanding of Amazon SimpleDB
      with PHP, Java, and Python examples, and run optimized
      database-backed applications on Amazon\\'s Web Services cloud",
    "icon": "http://static.ak.fbcdn.net/rsrc.php/zB010/hash/
      9yvl71tw.gif",
    "type": "link",
    "created_time": "2010-06-04T12:39:44+0000",
    "updated_time": "2010-06-04T12:39:44+0000",
    "likes": 1
}
```

That object has expired now, and is no longer available through the Graph API, but as you could have guessed, it was available at `https://graph.facebook.com/204603129458_127056137323572`. It's in the same format as the Page object – albeit with a few different fields – so our Graph Object Renderer could render it just fine.

Of course, this is useless unless we know the ID of each of the Posts associated with Packt Publishing, and there's no indication of where we might find them. Or is there?

I said earlier that the Graph API was designed to be self-documenting. We can request extra, "meta" information about any Graph Object by adding a `metadata=1` flag to the end of any Graph URL. Take a look at: `https://graph.facebook.com/PacktPub?metadata=1` in your browser. A new property, `type`, appears in the JSON:

```
"type": "page"
```

That's useful; as I said, Posts and Pages (and in fact all Graph Objects) take the same format, so this gives us a way of telling them apart.

More immediately interesting, though, is the new `metadata` object. This contains one object, `connections`, and one array, `fields`. Let's look at `fields` first:

```
"fields": [
 {
  "name": "id",
  "description": "The page's ID"
 },
 {
  "name": "name",
  "description": "The page's name"
 },
 {
  "name": "picture",
  "description": "The pages profile picture"
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
 },
 {
  "name": "category",
  "description": "The page's category"
 },
 {
  "name": "fan_count",
  "description": "\\* The number of fans the page has"
 }
]
```

This is a list explaining what each of the fields in the main body of the Graph Object represents. At time of writing, this is still a fairly new feature, so it's possible that the list will be more complete by the time you load it.

The `connections` object is as follows:

```
"connections": {
 "feed": "https://graph.facebook.com/packtpub/feed",
 "posts": "https://graph.facebook.com/packtpub/posts",
 "tagged": "https://graph.facebook.com/packtpub/tagged",
 "statuses": "https://graph.facebook.com/packtpub/statuses",
 "links": "https://graph.facebook.com/packtpub/links",
 "notes": "https://graph.facebook.com/packtpub/notes",
 "photos": "https://graph.facebook.com/packtpub/photos",
 "albums": "https://graph.facebook.com/packtpub/albums",
 "events": "https://graph.facebook.com/packtpub/events",
 "videos": "https://graph.facebook.com/packtpub/videos"
}
```

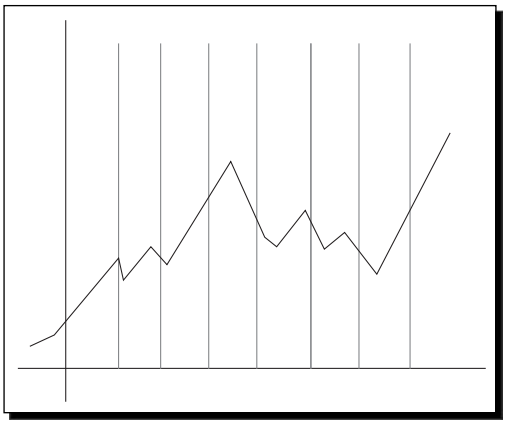
Browse to one of the URLs from the previous list: `http://graph.facebook.com/packtpub/posts`. It returns a JSON containing an array called `data` and an object called `paging`. The `data` array contains several Post objects; we'll look at `paging` later in the book.

What just happened?

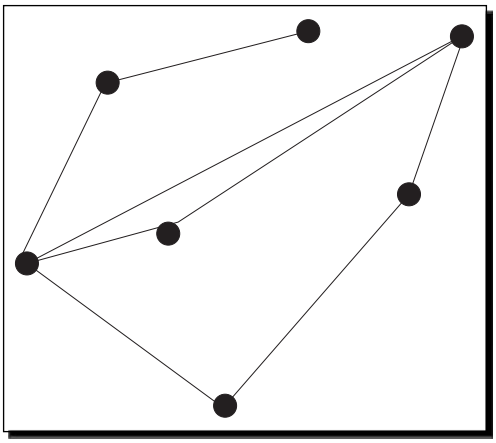
The `metadata=1` parameter tells the Graph API to display all of the metadata about the current object, which, in this case, includes the type of object it is, an array of descriptions of the object's properties, and all of the URLs that contain lists of objects connected to this Page.

This layout is where the Graph API gets its name. In everyday usage, "graph" means the type of chart shown in the next diagram:

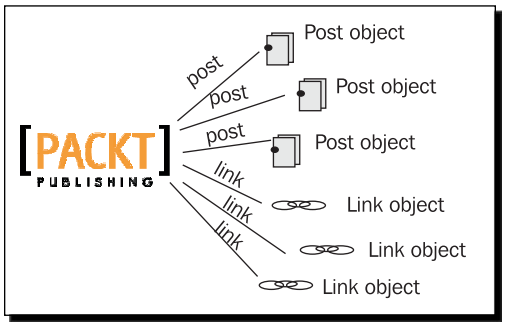
For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book



But in mathematics, "graph" refers to any set of nodes connected by edges, like the example in the next diagram:



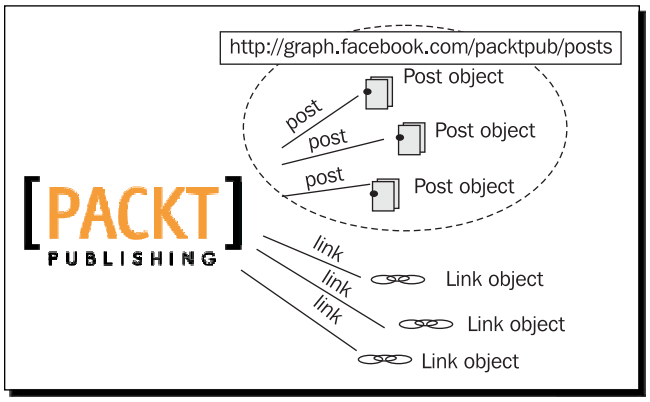
The Graph API represents Facebook's data as shown in the next diagram :



For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

In the previous diagram, each object is a node, and the lines represent different types of connection.

Fetching `http://graph.facebook.com/packtpub/posts` gets you all the nodes joined to `PacktPub` by a "`post`" connection – that is, all Post objects that have been posted on Packt's wall:



Have a go hero – exploring connections

Now that you know about the `metadata` parameter, explore the different types of connections in your browser, and see what new kinds of objects you can find.

Rendering Lists

What happens if you try to load `https://graph.facebook.com/packtpub/posts` using the same code we used to load the Packt Publishing Page object?

We get this in the output panel:

Graph Object was null!

Not a success. The way the Graph API structures the JSON here is totally different to how it structures the JSON for a Page, Post, or any other Graph Object. The same is true of the JSON for the other connection URLs. We call this structure a **Graph List**.

Time for action – rendering Lists of Posts

Since a Graph List's `data` property is an array of Graph Objects, we could just loop through the array and create a new Graph Object Renderer for each element. Feel free to have a go at this, if you like, but I've got another solution.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

I've created a second renderer: this time, a **Graph List Renderer**. I've also created a class `graph.GraphList`. And `CustomGraphContainerController` inherits a method called `renderGraphList()`. Perhaps unsurprisingly, this takes an object of type `graph.GraphList` as a parameter, and creates a new Graph List Renderer to display its contents. So, we need to take a Graph List that we receive from the Graph API, and turn it into an instance of the `GraphList` class. The `GraphList` class is a little more sophisticated than the `GraphObject` class; it has a method called `addToList()`, to which we can pass any `GraphObject` instance to be added to the list.

We'll still loop through the `data` array, then, but instead of rendering each `GraphObject` on its own, we'll add each one to a `GraphList` and render that.

Modify the URL that `CustomGraphContainerController` requests, so that it loads the list of posts:

```
public function CustomGraphContainerController
  (a_graphControlContainer:GraphControlContainer)
{
  super(a_graphControlContainer);

  var loader:URLLoader = new URLLoader();
  var request:URLRequest = new URLRequest();
  //Specify which Graph URL to load
  request.url = "https://graph.facebook.com/PacktPub/posts";
  loader.addEventListener(Event.COMPLETE, onGraphDataLoadComplete);
  //Start the actual loading process
  loader.load(request);
}
```

Now, once this is loaded, we need to check whether the item returned is a Graph Object or a Graph List. We can do this by looking for a property called `data`; if one exists, we'll assume it's a List.

```
private function onGraphDataLoadComplete(a_event:Event):void
{
  var loader:URLLoader = a_event.target as URLLoader;
  //obtain whatever data was loaded, and trace it
  var graphData:String = loader.data;
  var decodedJSON:Object = JSON.decode(graphData);

  if (decodedJSON.data)
  {
    //has a "data" property so we assume it is a Graph List

  }
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
    else
    {
      //no "data" so we assume it is a Graph Object
      var graphObject:GraphObject = new GraphObject();
      //copy all the properties from decodedJSON to graphObject
      for (var key:String in decodedJSON)
      {
        graphObject[key] = decodedJSON[key];
      }
      this.renderGraphObject(graphObject);
    }
  }
```

Inside this `if` block, we first create a new `GraphList` instance:

```
  if (decodedJSON.data)
  {
    //has a "data" property so we assume it is a Graph List
    var graphList:GraphList = new GraphList();
  }
```

(You will need to `import graph.GraphList`.)

Next, remember than `decodedJSON.data` is an array of objects; we loop through this array, and create a `GraphObject` from each element.

```
  if (decodedJSON.data)
  {
    //has a "data" property so we assume it is a Graph List
    var graphList:GraphList = new GraphList();
    var childGraphObject:GraphObject;
    for each (var childObject:Object in decodedJSON.data)
    {
      childGraphObject = new GraphObject();
      for (var childKey:String in childObject)
      {
        childGraphObject[childKey] = childObject[childKey];
      }
    }
  }
```

This is basically the same thing we did with the `decodedJSON` when loading a single Graph Object.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

What about the other property inside the Graph List, the `paging` object? We should add that too:

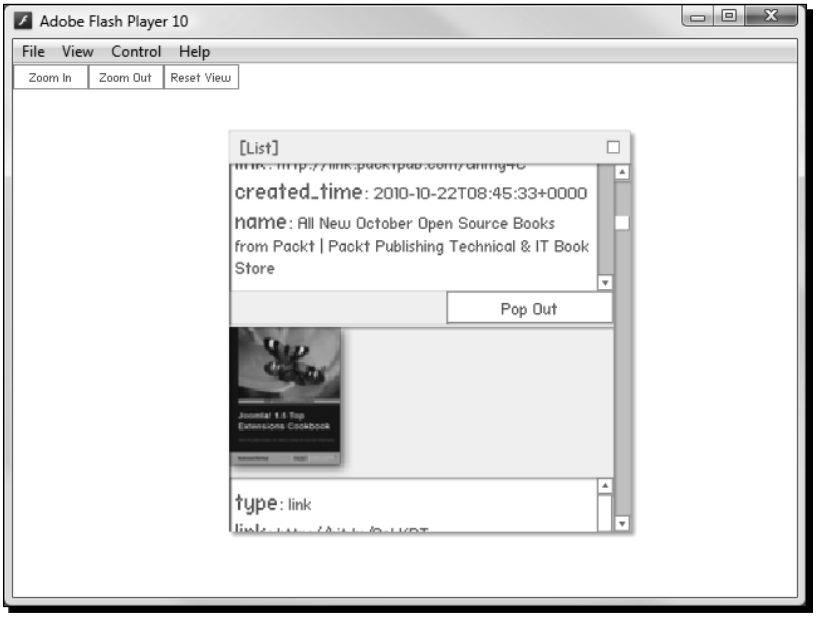
```
if (decodedJSON.data)
{
  //has a "data" property so we assume it is a Graph List
  var graphList:GraphList = new GraphList();
  var childGraphObject:GraphObject;
  for each (var childObject:Object in decodedJSON.data)
  {
    childGraphObject = new GraphObject();
    for (var childKey:String in childObject)
    {
      childGraphObject[childKey] = childObject[childKey];
    }
    graphList.addToList(childGraphObject);
  }
  graphList.paging = decodedJSON.paging;
}
```

Finally, we pass the `GraphList` instance to `renderGraphList()`:

```
if (decodedJSON.data)
{
  //has a "data" property so we assume it is a Graph List
  var graphList:GraphList = new GraphList();
  var childGraphObject:GraphObject;
  for each (var childObject:Object in decodedJSON.data)
  {
    childGraphObject = new GraphObject();
    for (var childKey:String in childObject)
    {
      childGraphObject[childKey] = childObject[childKey];
    }
    graphList.addToList(childGraphObject);
  }
  graphList.paging = decodedJSON.paging;
  this.renderGraphList(graphList);
}
```

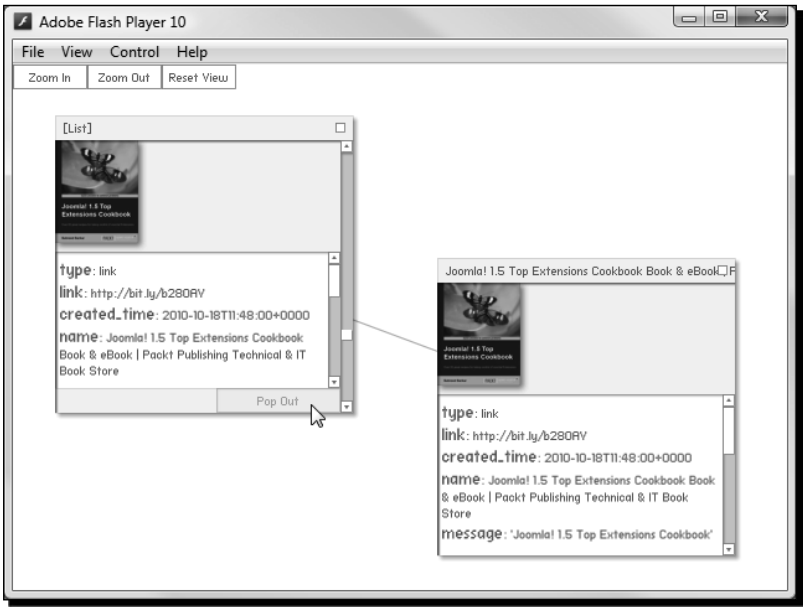

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Compile the SWF and test it. The following screenshot shows the result:



It's a scrollable window containing all the Graph Objects from the list.

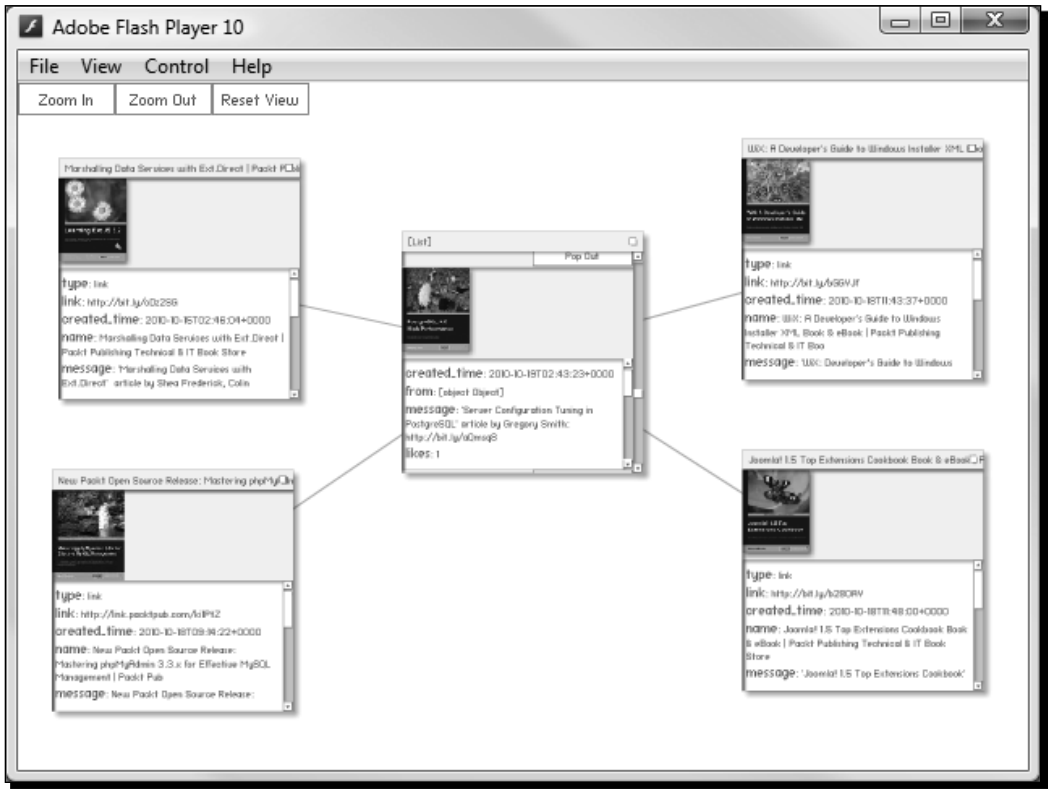
What happens when you click the **Pop Out** button underneath a Graph Object?



For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

What just happened?

The Graph Object pops out into its own Graph Object Renderer, with a gray line connecting it to the list to which it belongs. This lets you look at several children of a list at the same time:



(You can drag the individual renderers to reposition them, or drag the background to move everything at once.)

This makes it clear that a Graph List is just a collection of Graph Objects.

Rendering connections

We've shown the link from a Graph List to its Graph Objects; the next step is to show the connections from a Graph Object to its Graph Lists.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Time for action – displaying a Graph Object's connections

The Graph Object Renderer has the ability to show a list of all the object's connections, if that list is included as part of the Graph Object.

All we have to do is tell the Graph API to give us that list when we request a Graph Object; since our code for creating an instance of `GraphObject` from a JSON copies all the properties of that JSON to the `GraphObject`, this metadata will be included too. So, actually, all we need to do is add the `metadata=1` flag to the end of the Graph URL that we request, and it'll do the rest for us.

We could do this by changing our request code as shown in the following excerpt:

```
public function CustomGraphContainerController
  (a_graphControlContainer:GraphControlContainer)
{
  super(a_graphControlContainer);

  var loader:URLLoader = new URLLoader();
  var request:URLRequest = new URLRequest();
  //Specify which Graph URL to load
  request.url = "https://graph.facebook.com/PacktPub?metadata=1";
  loader.addEventListener(Event.COMPLETE, onGraphDataLoadComplete);
  //Start the actual loading process
  loader.load(request);
}
```

There's a slightly more elegant way to do this, however, using a class called `URLVariables`. In `CustomGraphContainerController.as`, add a line to import this class:

```
import flash.net.URLVariables;
```

Now, modify the constructor as shown in the following lines of code:

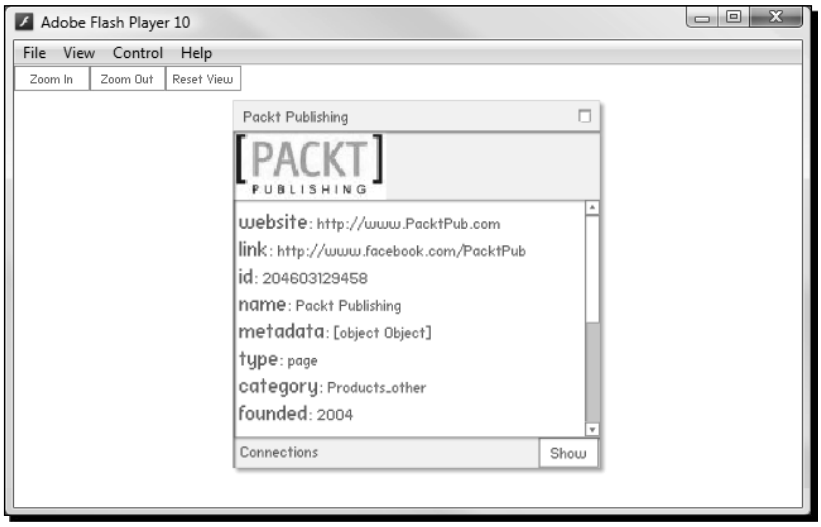
```
public function CustomGraphContainerController
  (a_graphControlContainer:GraphControlContainer)
{
  super(a_graphControlContainer);

  var loader:URLLoader = new URLLoader();
  var request:URLRequest = new URLRequest();
  var variables:URLVariables = new URLVariables();
  //Specify which Graph URL to load
  request.url = "https://graph.facebook.com/PacktPub";
  variables.metadata = 1;
  request.data = variables;
  loader.addEventListener(Event.COMPLETE, onGraphDataLoadComplete);
  //Start the actual loading process
  loader.load(request);
}
```

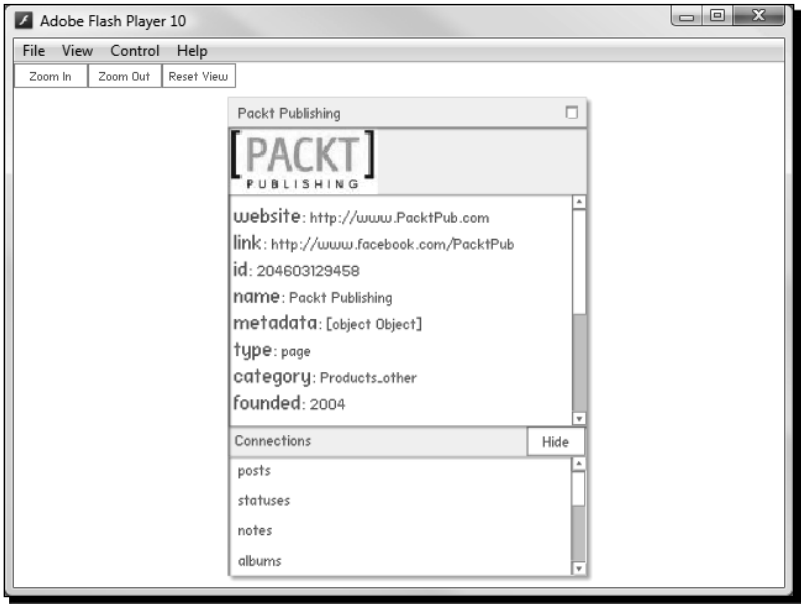

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

As you can probably guess, setting `variables.metadata = 1` is exactly the same as sticking `?metadata=1` on the end of the URL. Doing it this way takes a few more lines, but it makes it much easier to set different parameters, and keeps the parameters separate from the URL.

Anyway, compile the SWF and you should see the following screenshot:



Notice the new **Connections** bar at the bottom of the Renderer? Click on the **Show** button:



For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

What just happened?

We can now see all of the connections of a Graph Object right there in its renderer. Of course, that's not very interesting unless we can see what's at the other end of each connection!

Introducing the Requestor

Ideally, whenever the user clicks a connection from the scrolling list, a new Graph List Renderer of that connection will be created and displayed.

To do this, we'd need to add a `MouseEvent.CLICK` listener to the list, and use it to trigger a new `URLLoader` request for the clicked connection.

Fortunately, all the UI code has already been provided elsewhere in the project; we just need to tap into that. To do this, we'll need to make use of what I call a **Requestor**.

Time for action – creating an HTTP Requestor

The idea is, we move all of the code regarding the `URLLoader` from `CustomGraphContainerController` to a separate class, called `HTTPRequestor`. We will then replace the `CustomGraphContainerController` constructor with this:

```
public function CustomGraphContainerController(a_graphControlContainer
:GraphControlContainer)
{
  super(a_graphControlContainer);

  _requestor = new HTTPRequestor();
  _requestor.request(new GraphRequest("PacktPub"));
}
```

Why bother? Well, apart from being neater, there are two main advantages:

1. It's much simpler to request several Graph Objects or Graph Lists; no need to deal with multiple instances of `URLLoader`.
2. In the next chapter, we'll see how to use the official Adobe AS3 Facebook SDK to retrieve information from the Graph API. If all the code for a request is encapsulated in one class, then we only need to change one line to switch from using HTTP to using Adobe's SDK:

   ```
   public function CustomGraphContainerController(a_graphControlConta
   iner:GraphControlContainer)
   {
     super(a_graphControlContainer);

     _requestor = new SDKRequestor();
     _requestor.request(new GraphRequest("PacktPub"));
   }
   ```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book



`GraphRequest` is a simple class; its constructor allows you to use two parameters to specify what you'd like to retrieve from the Graph API:

- `objectID`, the name of any Graph Object.
- `connectionID`, the name of any connection of that Graph Object.

So, to request the Packt Publishing Page, you would use this `GraphRequest`:

```
newGraphRequest("PacktPub");
```

and to request the list of Posts from the Packt Publishing Page, you'd use this:

```
newGraphRequest("PacktPub", "posts");
```

The class is already written; it's in `\src\com\graph\apis\http\HTTPRequestor.as`. Take a look! There are a few changes from the code we wrote in `CustomGraphContainerController.as`, but these all have comments to explain them:

```
package graph.apis.http
{
  import events.DialogEvent;
  import events.RequestEvent;
  import flash.events.Event;
  import flash.events.EventDispatcher;
  import flash.events.HTTPStatusEvent;
  import flash.events.IEventDispatcher;
  import flash.events.IOErrorEvent;
  import flash.net.URLLoader;
  import flash.net.URLRequest;
  import flash.net.URLVariables;
  import flash.utils.Dictionary;
  import graph.apis.base.IRequestor;
  import graph.BaseGraphItem;
  import graph.GraphList;
  import graph.GraphObject;
  import graph.GraphRequest;
  import com.adobe.serialization.json.JSON;

  //the class needs to dispatch events (see later in code for why)
  public class HTTPRequestor extends EventDispatcher implements
    IRequestor
  {
    //this is used to figure out which GraphRequest created each
    //loader
    private var _requests:Dictionary = new Dictionary();

    public function HTTPRequestor(target:IEventDispatcher = null)
    {
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
        //this is needed because the class extends EventDispatcher
        super(target);

    }

    public function request(a_request:GraphRequest):void
    {
      var loader:URLLoader = new URLLoader();
      var urlRequest:URLRequest = new URLRequest();
      var variables:URLVariables = new URLVariables();

      //We construct a URL from the parameters of the GraphRequest
      urlRequest.url = "https://graph.facebook.com/" + a_request.
objectID;
      if (a_request.connectionID)
      {
        urlRequest.url += "/" + a_request.connectionID;
      }

      variables.metadata = 1;
      urlRequest.data = variables;

      //this is used to figure out which GraphRequest created the
loader later
      _requests[loader] = a_request;

      loader.addEventListener(Event.COMPLETE,
onGraphDataLoadComplete);
      loader.load(urlRequest);
    }

    private function onGraphDataLoadComplete(a_event:Event):void
    {
      var loader:URLLoader = a_event.target as URLLoader;
      var graphData:String = loader.data;
      var decodedJSON:Object = JSON.decode(graphData);

      //we find the original GraphRequest used to start the loader
      var originalRequest:GraphRequest = _requests[loader] as
GraphRequest;

      if (decodedJSON.data)
      {
        var graphList:GraphList = new GraphList();
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
            var childGraphObject:GraphObject;
            for each (var childObject:Object in decodedJSON.data)
            {
              childGraphObject = new GraphObject();
              for (var childKey:String in childObject)
              {
                childGraphObject[childKey] = childObject[childKey];
              }
              graphList.addToList(childGraphObject);
            }
            graphList.paging = decodedJSON.paging;

            //we use the properties of the original GraphRequest to add
            //some extra data to the GraphList itself
            graphList.ownerID = originalRequest.objectID;
            graphList.connectionType = originalRequest.connectionID;

            //since this class does not have a renderGraphList() method,
            //we dispatch an event, which CustomGraphContainerController
            //will listen for, and call its own renderGraphList() method
            dispatchEvent(new RequestEvent(RequestEvent.REQUEST_COMPLETED,
              graphList));
          }
          else
          {
            var graphObject:GraphObject = new GraphObject();
            for (var key:String in decodedJSON)
            {
              graphObject[key] = decodedJSON[key];
            }

            //since this class does not have a renderGraphList() method,
            //we dispatch an event, which CustomGraphContainerController
            //will listen for, and call its own renderGraphList() method
            dispatchEvent(new RequestEvent(RequestEvent.REQUEST_COMPLETED,
              graphObject));
          }
        }
      }

    }
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

There's no need to change any of this, or even to understand any of it apart from the HTTP request code that we wrote earlier. Just remember, its purpose is to encapsulate your requests to the Graph API.

Now, go back to `CustomGraphContainerController.as` and remove all the request-related code:

```
package controllers
{
  import ui.GraphControlContainer;

  public class CustomGraphContainerController extends GCController
  {

    public function CustomGraphContainerController
      (a_graphControlContainer:GraphControlContainer)
    {
      super(a_graphControlContainer);

    }

  }

}
```

`CustomGraphContainerController` inherits a protected variable called `_requestor` of type `IRequestor`, as well as a method for adding the required event listeners to it, so all we need to do is this:

```
package controllers
{
  import graph.apis.http.HTTPRequestor;
  import graph.GraphRequest;
  import ui.GraphControlContainer;

  public class CustomGraphContainerController extends GCController
  {

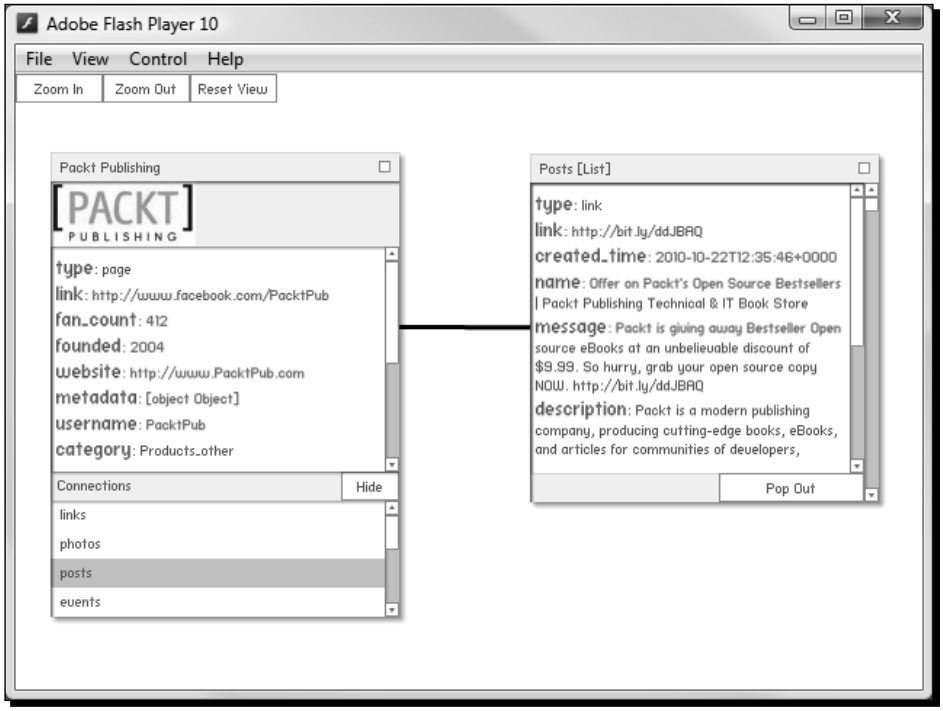
    public function CustomGraphContainerController
      (a_graphControlContainer:GraphControlContainer)
    {
      super(a_graphControlContainer);
      _requestor = new HTTPRequestor();
      addEventListenersToRequestor();
      _requestor.request(new GraphRequest("PacktPub"));
    }

  }

}
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Compile and run your SWF, then expand the **Connections** box and click on "**posts**":



Great! The Graph List Renderer appears, with a black line to the Page to indicate that there is a connection between them. What about the other connections? Try clicking on **statuses**.

Error #2044: Unhandled ioError:.text=Error #2032: Stream Error. URL: https://graph.facebook.com/204603129458/statuses?metadata=1

Oops.

What just happened?

If you load the troublesome URL in your browser (`https://graph.facebook.com/packtpub/statuses`), you'll see the following message:

```
{
   "error": {
      "type": "OAuthAccessTokenException",
      "message": "An access token is required to request this
        resource."
   }
}
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

This error is due to not being logged in to Facebook through your SWF. We'll look at how to solve this in the next chapter.



For now, you can get around the error by adding an `IO_ERROR` event listener to the `URLLoader`. In `HTTPRequestor.as`, modify `request()`:

```
public function request(a_request:GraphRequest):void
{
  varloader:URLLoader = new URLLoader();
  varurlRequest:URLRequest = new URLRequest();
  varvariables:URLVariables = new URLVariables();

  //We construct a URL from the parameters of the
  //GraphRequest
  urlRequest.url = "https://graph.facebook.com/" +
      a_request.objectID;
  if (a_request.connectionID)
  {
    urlRequest.url += "/" + a_request.connectionID;
  }

  variables.metadata = 1;
  urlRequest.data = variables;

  //this is used to figure out which GraphRequest
  //created the loader later
  _requests[loader] = a_request;

  loader.addEventListener(Event.COMPLETE,
    onGraphDataLoadComplete);
  loader.addEventListener(IOErrorEvent.IO_ERROR,
    onIOError);
  loader.load(urlRequest);
}
```

You will need to import `flash.events.IOErrorEvent`. Now, in the same class, create a simple event handler function to trace the error:

```
private function onIOError(a_event:IOErrorEvent):void
{
  trace(a_event.text);
}
```

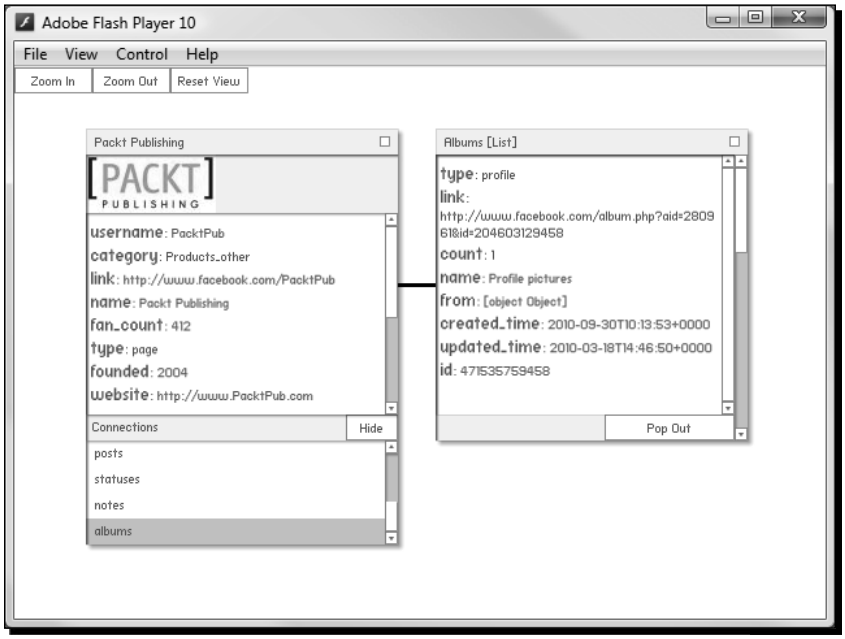


This way, you can see the error in your output window, but it won't crash the SWF. Note: a try-catch block will not work for this kind of error.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Understanding Connections of Connections

Take a look at the Graph List Renderer created by clicking on the "**album**" connection:



Notice anything missing?

There are no pictures! We can see lots of photos when loading the Packt Publishing Page inside the actual Facebook website, but here there are no photo URLs at all. Check it out by loading the Graph List in the browser; even with `?metadata=1`, there's no indication of where the photos might be:

```
{
   "data": [
      {
         "id": "471535759458",
         "from": {
            "name": "Packt Publishing",
            "category": "Products_other",
            "id": "204603129458"
         },
         "name": "Profile pictures",
         "link": "http://www.facebook.com/
           album.php?aid=280961&id=204603129458",
         "count": 1,
         "type": "profile",
         "created_time": "2010-09-30T10:13:53+0000",
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
            "updated_time": "2010-03-18T14:46:50+0000"
         },
         {
            "id": "307932939458",
            "from": {
               "name": "Packt Publishing",
               "category": "Products_other",
               "id": "204603129458"
            },
            "name": "Books",
            "description": "Packt Books",
            "link": "http://www.facebook.com/
              album.php?aid=180619&id=204603129458",
            "count": 32,
            "type": "normal",
            "created_time": "2010-02-04T12:32:17+0000",
            "updated_time": "2010-03-18T16:08:42+0000"
         }
      ],
      "paging": {
         "previous": "https://graph.facebook.com/204603129458/
        albums?metadata=1&limit=25&since=2010-09-30T10%3A13%3A53%2B0000",
         "next": "https://graph.facebook.com/204603129458/
   albums?metadata=1&limit=25&until=2010-02-04T12%3A32%3A16%2B0000"
      }
   }
```

Time for action – loading photos from an album

However, as we've established, each object inside `data` is a Graph Object in its own right. Let's take a look at the Packt Books album, with id `307932939458`, by browsing to `https://graph.facebook.com/307932939458?metadata=1`:

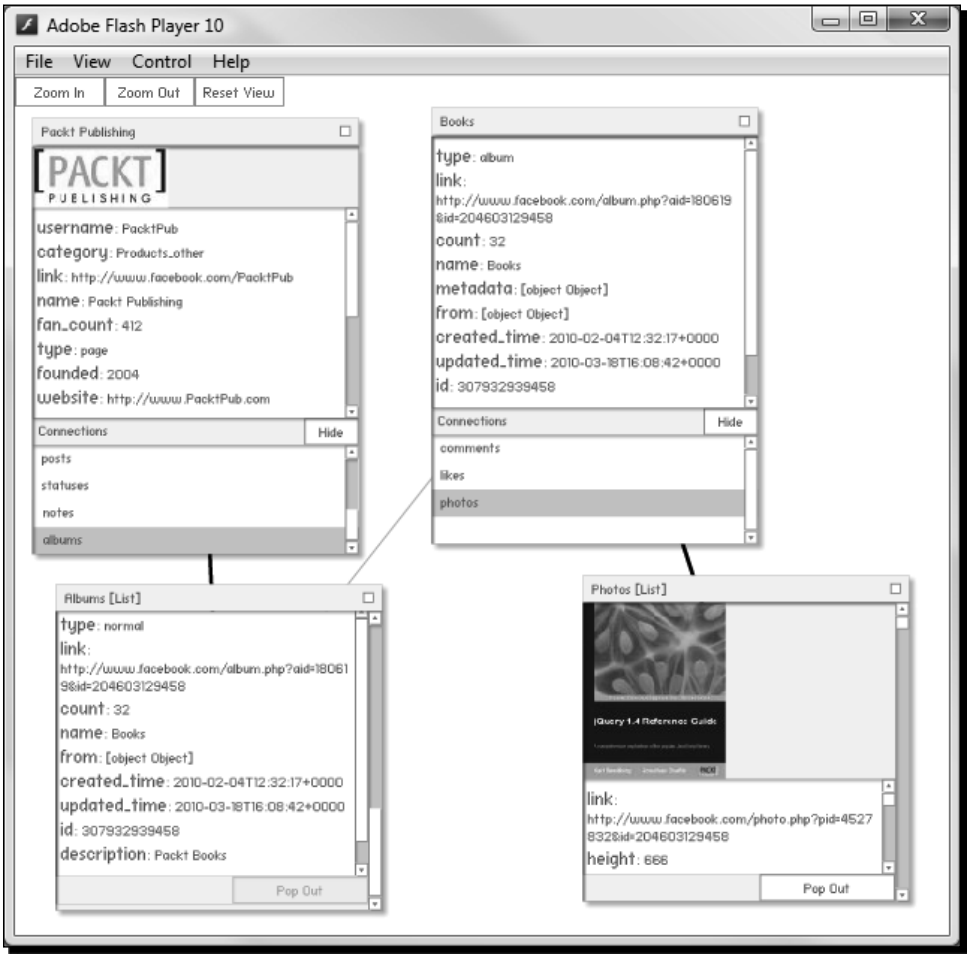
```
{
   "id": "307932939458",
   "from": {
      "name": "Packt Publishing",
      "category": "Products_other",
      "id": "204603129458"
   },
   "name": "Books",
   "description": "Packt Books",
   "link": "http://www.facebook.com/
     album.php?aid=180619&id=204603129458",
   "count": 32,
   "type": "album",
   "created_time": "2010-02-04T12:32:17+0000",
   "updated_time": "2010-03-18T16:08:42+0000",
   "metadata": {
```


For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

```
        "connections": {
           "photos": "https://graph.facebook.com/307932939458/photos",
           "likes": "https://graph.facebook.com/307932939458/likes",
           "comments": "https://graph.facebook.com/307932939458/
             comments"
        },
     }
  }
```

This time, the metadata gives us the information we need. The photos are linked to the Album Graph Object through a connection called "photos".

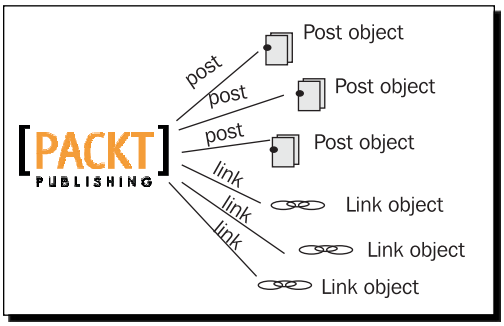
Run your SWF and load the **albums** connection again. In the Renderer, scroll to the Graph Object whose name is Books, and click on **Pop Out**. Then, expand the Connections box of the **Books Renderer**, and click on **photos**.



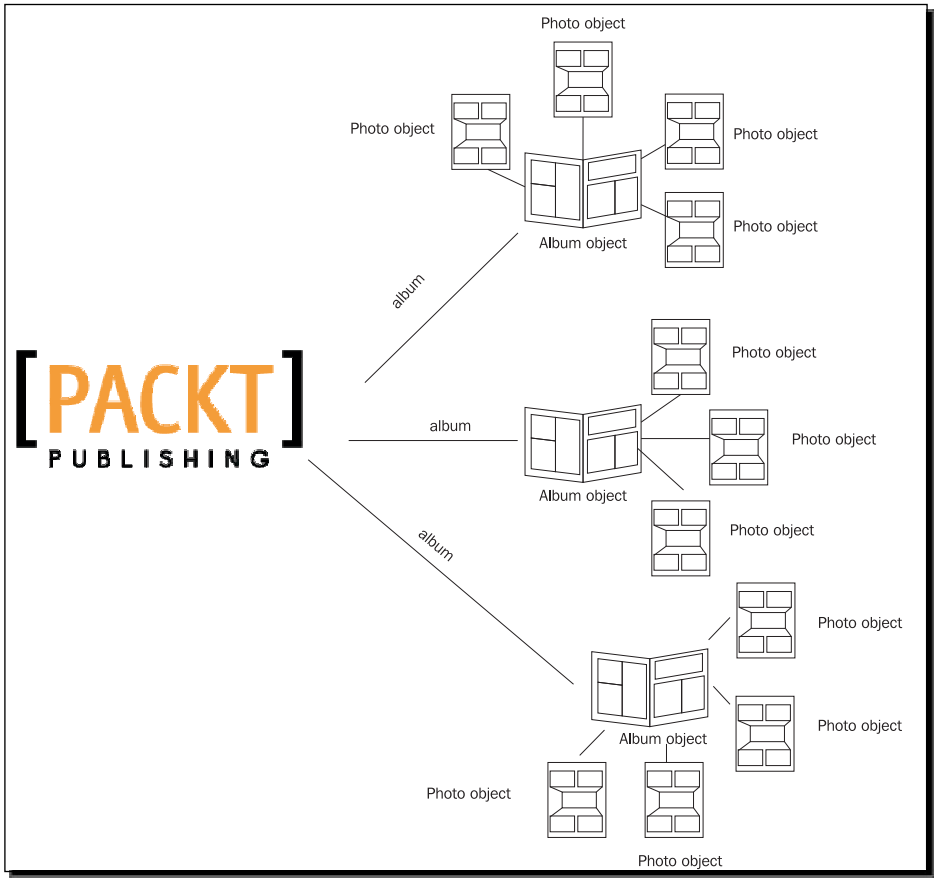
For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

What just happened?

When we were only considering the "posts" connection, our graph was very simple; there was a single connection between the Page and everything related to it:



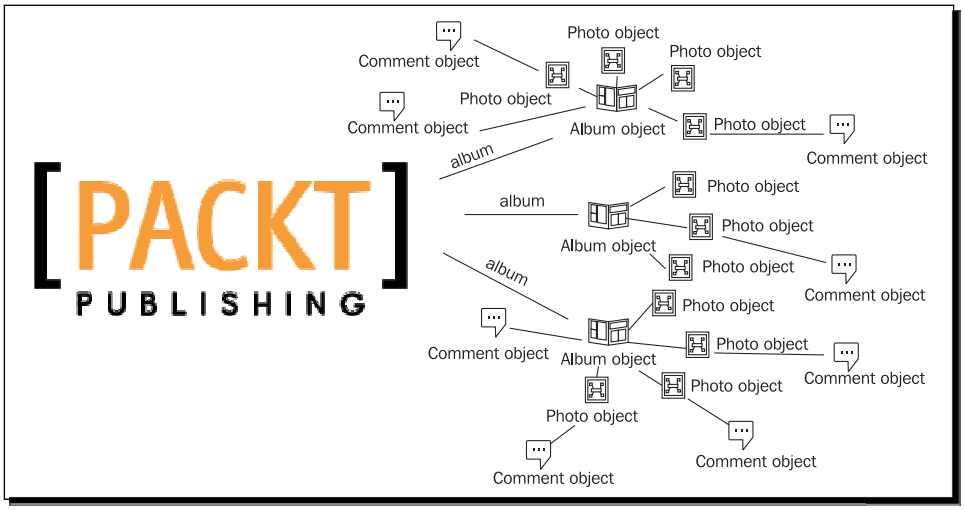
Now that we've introduced albums, it's more complicated:



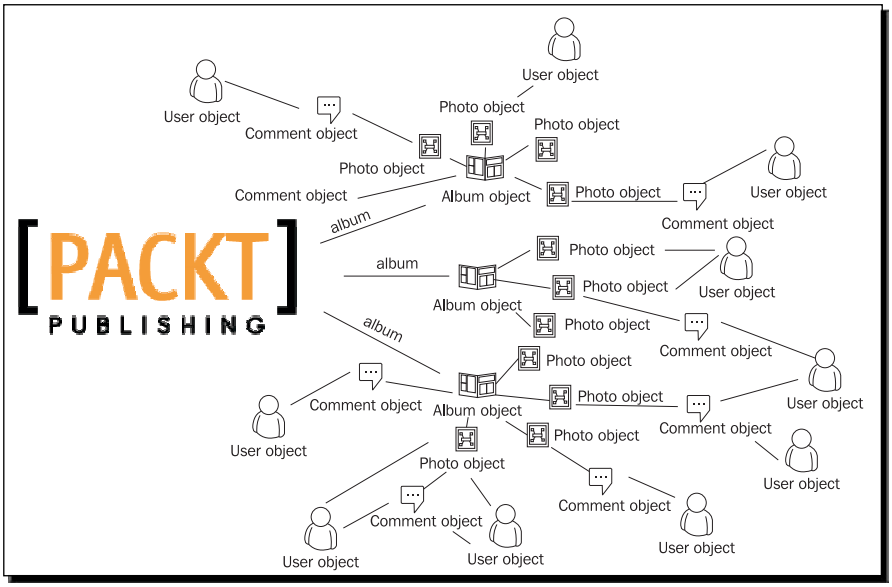
For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

We now have to traverse two levels of connection to get from the Page to the objects we're looking for.

The connections don't stop there, though. Both Albums and Photos can be connected to Comments, too:

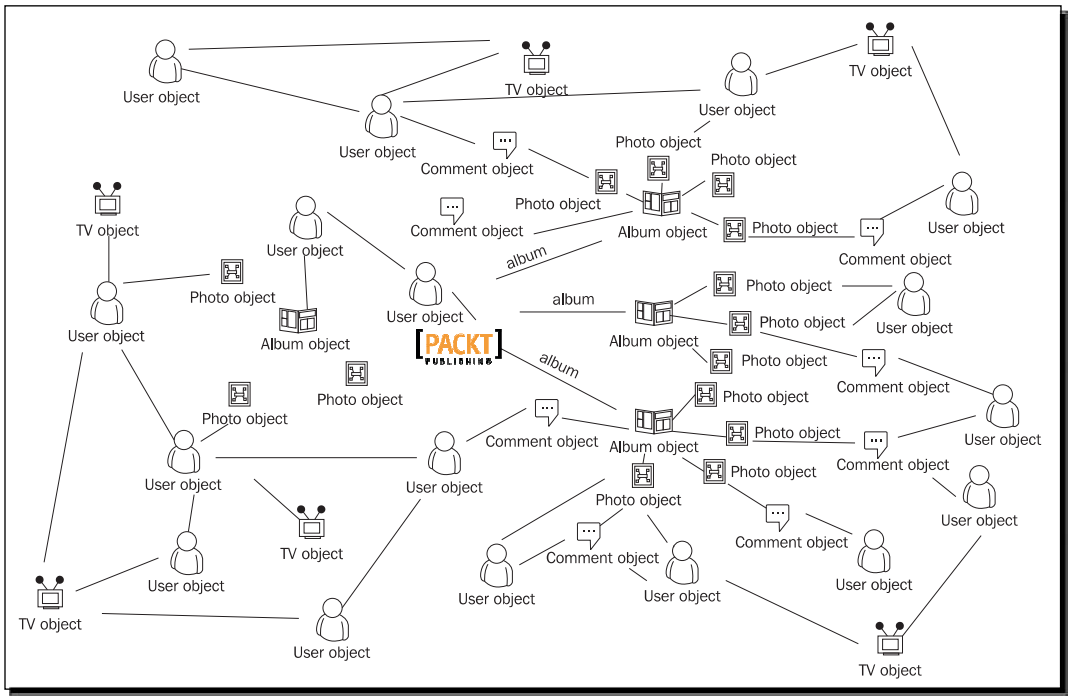


Plus, every comment has a property called `from` that connects it to the user that posted it. A user can also be "tagged" as appearing in a photo, which connects the photo and the user, as well:



For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

The diagram is looking more and more complex (and the similarities to the mathematical graph drawn earlier are now clear). Of course, now that people are involved, the number of connections gets ridiculous. Users can be connected to any other object, either by being friends with another User, by being tagged in a Photo, Video, or Note, by posting a Comment, Link or other item, or by clicking **Like** on any other element in Facebook.



If you start with one Page and keep going through all the objects connected to it, and then all the objects connected to those, and so on, you can cover huge numbers of nodes without having to start again with a new Page.

The power of the graph lies in its flexibility. Every type of Graph Object has the same basic structure as every other type of Graph Object – with the exception of Graph Lists (which contain arrays of Graph Objects). That's why our Graph Object Renderer can easily display any kind of Graph Object.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Also, have you noticed that it's not just comments that have a `from` property? Albums do, too, and so do individual photos, and pretty much every type of object that isn't a Page or a User. This means you can start with any object, find its creator, and traverse outwards across the graph from there.

The Graph model has implications for privacy. Suppose that, if we were granted access to information about a Page, we were also allowed to access information about any object connected to that Page. Well, then we could go from the Page to:

- An Album posted by the Page, to
- A Photo in that Album, to
- A User tagged in that Photo, to
- That User's list of wall posts, to
- A Comment made on a post by a friend of the first User, to
- The User that posted the Comment, to
- A TV Show that this User Likes, to
- A Link posted on the Page for that TV Show, to
- The User who posted that link

and so on. It's no surprise, then, that Facebook uses a more detailed set of rules to determine both what a user can access, and what an app can access on behalf of a user. We'll look at these rules in the next chapter.

Putting it all together

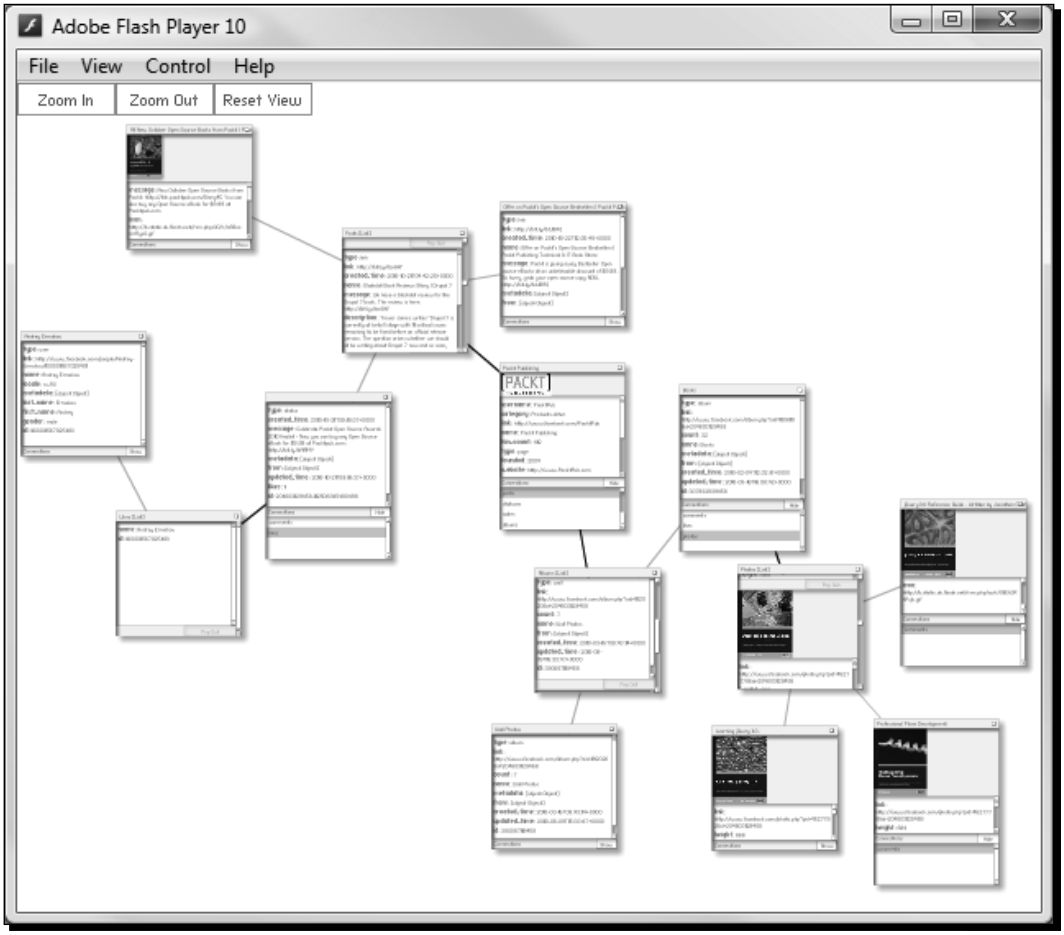
Finally, let's see how far we can traverse through the graph, starting from the Packt Publishing Page.

Time for action – traversing the Graph

Set the Visualizer to start by requesting the PacktPub Page. Now, compile and run your SWF, and use the **Connections** box and the **Pop Out** buttons to explore the Graph, and see how far you can get. Don't forget you can drag Renderers around, and zoom out to fit even more in the Flash Player window! And remember, black lines signify connections, while gray lines signify that the object belongs to a List.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

The following screenshot shows what it could look like after only a few clicks:



You can already see the resemblance to the sprawling diagrams of the Graph seen previously in the chapter.

What just happened?

You've written the code to power an RIA that allows you to explore the entire public Graph, starting from any point. In other words, you've made a Facebook crawler, in Flash.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Have a go hero – exploring other areas

You don't have to start exploring from the PacktPub Page. Try changing that initial `GraphRequest` instance to request the Facebook Page, `Facebook`, or *Mark Zuckerberg's* public profile, `markzuckerberg`, or the Page of any other brand, company, or famous person.

Also, realize that you're not limited to a single `GraphRequest`; you can create as many as you like. Try starting with a few at once, and see if you come across any overlaps!

Keep an eye on your output window for traces telling you that a Graph Object or List could not be retrieved. Is it always for the same reason?

Pop Quiz

1. What does the `?metadata=1` parameter do when used in a Graph API URL?
    a. It makes the metadata visible
    b. It makes the metadata invisible
2. How many levels through the Graph can you traverse, starting with any Page?
    a. One
    b. Two
    c. Ten
    d. Unlimited
3. True or false: If the JSON returned from a Graph URL contains an object called data, we can always assume it's a Graph List.
    a. True
    b. False
4. True or false: If the JSON returned from a Graph URL doesn't contain an object called data, we can always assume it's a Graph Object.
    a. True
    b. False

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Summary

We learned a lot in this chapter about the Graph API: not just what it is, but also how to access it in AS3.

Key things to learn:

- The Graph API is so-called because it represents all of Facebook's data connected in an enormous graph of objects and connections.
- The Graph API has two types of elements: Graph Objects and Graph Lists.
- Graph Objects may have two IDs: a numeric one specified by Facebook, and possibly an alphanumeric one, specified by the Graph Object's owner.
- Graph Objects have connections; connections lead to Graph Lists; Graph Lists contain Graph Objects.
- The format of a Graph URL for retrieving a Graph Object is `https://graph.facebook.com/graph_object_id`.
- The format of a Graph URL for retrieving a Graph List is `https://graph.facebook.com/graph_object_id/connection_id`.
- Graph URLs return data in JSON format. This is a text-based format which uses key-value pairs to represent objects containing properties, arrays, and other objects.
- Sometimes, Graph URLs return error messages; these are also given in JSON format.
- We can use the `?metadata=1` parameter in a Graph URL to make it return extra information about the element, like the list of connections leading from a Graph Object.
- Metadata is not returned for Graph Objects that are part of a Graph List.
- The JSON representation of a Graph Object can be deserialized to an AS3 object we can use in code using the **as3corelib** library.

We also discussed how the Graph API is so flexible, because it uses the same basic structure for every type of object in Facebook's database.

But what about those objects that give us an authorization error when we try to get information on them? That's what we'll cover in the next chapter.

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book

Where to buy this book

You can buy Facebook Graph API Development with Flash Beginner's Guide from the Packt Publishing website: `https://www.packtpub.com/facebook-graph-api-development-with-flash/book`

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.



www.PacktPub.com

For More Information:
www.packtpub.com/facebook-graph-api-development-with-flash/book