Chris Cole
Chad Russell
Jessica Whyte

Foreword by
Dan Peterson, President, OpenSocial Foundation

Building OpenSocial Apps

A Field Guide to Working with the MySpace Platform

Developer's Library

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The screenshots and other depictions of myspace.com contained in this book may not accurately represent myspace.com as it exists today or in the future, including without limitation with respect to any policies, technical specs or product design.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Cole, Chris, 1974-
Building OpenSocial apps : a field guide to working with the MySpace platform/Chris Cole, Chad Russell, Jessica Whyte.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-61906-8 (pbk. : alk. paper)
ISBN-10: 0-321-61906-4 (pbk. : alk. paper)
1. Entertainment computing. 2. Internet programming. 3. MySpace.com. 4. OpenSocial.
5. Web site development. 6. Social networks—Computer network resources.
7. Application program interfaces (Computer software) I. Russell, Chad. II. Whyte, Jessica. III. Title.

QA76.9.E57C65 2010
006.7'54—dc22

2009032342

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-61906-8
ISBN-10: 0-321-61906-4

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

First printing, October 2009

Editor-in-Chief
Mark L. Taub

Acquisitions Editor
Trina MacDonald

Development Editor
Songlin Qiu

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Project Management
diacriTech LLC

Copy Editor
Barbara Wood

Indexer
Jack Lewis

Proofreader
George Seki

Technical Reviewers
Cassie Doll
Bess Ho
Benjamin Schupak

Book Designer
Gary Adair

Compositor
diacriTech LLC

Foreword

The Internet is constantly evolving, with vast arrays of information on every topic growing at a remarkable pace. Google's 1998 search index had only 26 million Web pages; a decade later it recognizes more than 1 trillion URLs. With so much information, we need a new set of tools to make sense of the Internet. The transition is from a strict focus on informational content, to being able to take advantage of our context, our relationships, and our activities.

Enter the social Web, a relatively new twist to the Internet, which is being built up by social networks, portals, and even more traditional businesses. The "Open Stack" is a set of specifications being developed by grass-roots communities all over the world, enabling developers to create new products and services enhanced by user-specific data. The Open Stack includes specifications such as OAuth, which provides secure access to data; OpenID, a global identity standard; and OpenSocial, a common API for building applications. These specifications are becoming the underlying infrastructure for the social Web, weaving a social fabric throughout the Web.

OpenSocial enables developers to learn a single core programming model that can be applied to all "OpenSocial containers," those sites that support the OpenSocial specification. With standards-based tools, including a JavaScript-based gadget API, a REST-based data access API, lightweight storage capabilities, and access to common viral channels, developers can build inside those containers as well as create applications for mobile phones or other sites. In late 2009, less than two years after its introduction, more than 50 sites have implemented support for the OpenSocial specification. In aggregate, these sites provide developers with access to more than 750 million users all over the world.

By taking advantage of the OpenSocial API and the information available in this book, you will have a great opportunity to reach a lot of users. For example, you'll find the OpenSocial API supported by many major sites that span the globe: Yahoo!, iGoogle, Xiaonei and 51.com (China), Mixi (Japan), orkut (Brazil), Netlog (Europe), and, of course, MySpace. Beyond that, OpenSocial is also supported by more productivity-oriented sites like LinkedIn and by Fortune 100 companies as diverse as IBM and Lockheed Martin.

With this book, you can quickly get up and running with OpenSocial on MySpace, and you'll be poised to leverage that experience to reach users on other sites as well. The in-depth programming examples provide a good introduction to the design options available when building with the OpenSocial API, and the code is open source for ease of use and future reference. Additionally, the MySpace Platform tips sprinkled throughout will help you avoid common mistakes and understand the intricacies of their platform policies. Getting a feel for how social platforms operate will be valuable as you continue to explore the wide world of OpenSocial.

OpenSocial is constantly evolving, just like the rest of the Internet. The OpenSocial specification is managed through an open process where anyone can contribute their ideas to influence the next version of the specification and help move the social Web forward.

Since its creation, there have been several significant revisions to the specification, introducing some new programming methodologies and improvements that make it easier for new developers to start using OpenSocial. As you're getting into the OpenSocial API, be sure to contribute back to the OpenSocial.org community your ideas on how to improve the specification.

It's open. It's social. It's up to you.

—Dan Peterson, president, OpenSocial Foundation
San Francisco, California
August 2009

Introduction

Welcome to the wonderful world of social apps. You are about to enter—or have already entered—a fast-paced and treacherous landscape that can be exciting, frustrating, intellectually challenging—and yes, it can even be pretty profitable.

We hope to be your guide through both the hidden pitfalls and the soaring peaks. There is much to gain from this exciting new market; social apps have been around for only a few years, after all. Many successful apps have yet to be created and then discovered by the millions upon millions of passionate MySpace users.

In this book we'll start from scratch and walk you through the entire process of building apps, from signing up as a developer all the way to building highly complex social apps that can scale out to thousands of users. We have extensive experience on the MySpace Development Platform (MDP), from building apps to helping build the platform itself. We'll point out the many idiosyncrasies and quirks of the platform, as well as the many ways to squeeze out a little better performance by making a few tweaks to your existing apps.

Throughout this book we'll demonstrate best practices and show lots of sample code as we take a step-by-step approach to app development. Starting with a simple "Hello World"–style app, we'll add functionality until we have a fully built, feature-rich app. We'll fetch data on the current user, get the user's friend list and photos, and parse all the data that comes back to display it on the screen. We'll send app invitations and notifications to help spark viral growth and send requests back to Web services running on a third-party server in order to process and store data. Finally, and possibly most important, we'll show you how to make money developing apps using ads and micropayments.

When it comes to developing social apps on the MySpace platform, the sky is the limit. As of this writing, the most popular app on MySpace has more than 13 million installs. Maybe your app will be the next big thing. All it takes is a good idea and a little bit of knowledge. If you provide the former, we can provide the latter.

Our hope is that this book is accessible both to experienced social app developers and to those developers who have only heard about it on the news. To that end, if you're standing in the technology section of your local bookstore reading this, and you have a bit of computer programming knowledge mixed in with some free time and curiosity, this book is for you.

We'll start from scratch in this introduction to ramp you up on MySpace and OpenSocial. For those who already have apps up and running out in the wild, this book is also for you, as we'll dig pretty deeply into the platform a little later on. You may be familiar with the ideas and terms we introduce here, so feel free to skip ahead a bit. (Check out the section on tools, though!)

How to Read This Book

There are a couple of ways to read this book. If you're an experienced app developer, you can use this as a reference book. The index will direct you to specific topics, where you can consult the various tables, sample code, and tips. Or, if you need to learn a new aspect of the platform, you can skip directly to that chapter and dive right in.

If you're not an experienced app developer, we suggest you start from the beginning. We'll start off with a "Hello World"–style app in Chapter 1 and add to it in each subsequent chapter. By the time the app is completed, you'll have a broad knowledge base of the entire platform. We then branch out into some advanced topics to take you a step further.

Assumptions Made by the Authors

This is not a book about how to code in JavaScript or Python or any other language. We will try to follow good practice in our sample code throughout the book, and we'll point out a few interesting tidbits along the way, but that's all. This is a book on how to write OpenSocial apps on MySpace.

To that end, we assume the following:

- You have basic knowledge of computer programming in general. Maybe you work at a software company and write PHP every day, or you've taken a few programming classes, or you've taught yourself by constructing your own Web page.
- You have basic knowledge of JavaScript or are willing to pick up the basics. We're not talking advanced stuff here, just the basics like calling functions, giving values to variables, and that kind of thing.

With all that said, let's get started!

What Is MySpace?

Ah, MySpace. Land of lost friends now found, hours wasted, and every band on the planet, large and small. Oh, and spam. Lots of spam. You can find the site here: www.myspace.com.

MySpace is a social network. You sign up for free and you're given a "profile"; the Profile page is a Web page that others can view and you can edit. On your Profile you can add information about yourself, pick a display name, and upload a picture. Others who view your Profile can leave comments for all to see. The cascading style sheets (CSS) of Profile pages can also be edited, so individuals can alter the styles of the page to "bling" out their Profiles in ways both beautiful and stunningly, shockingly bad.

In addition to a Profile page, you are given a Home page (shown in Figure I.1), which only you can see. It contains your notifications, "You have new mail!" and the like, along with various links to other parts of the site and updates from your friends.

It is the friend aspect that truly makes a social network. The idea is simple; you sign up and are assigned a single solitary friend: Tom, one of the cofounders of MySpace.

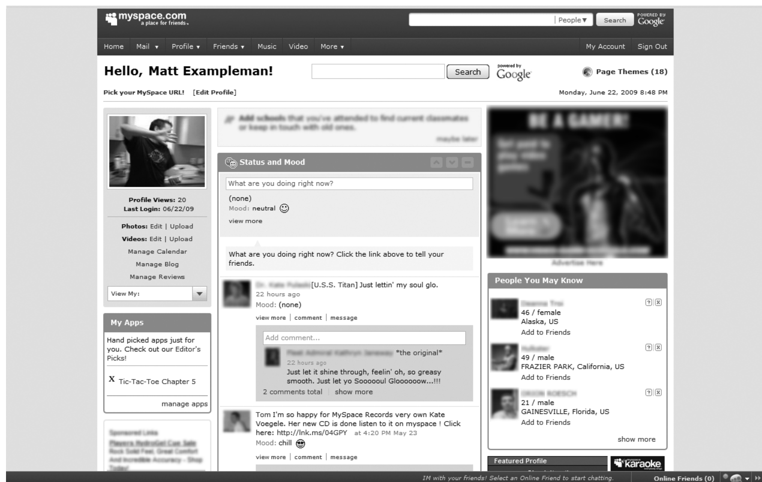

Figure I.1 A MySpace Home page.

Through various means, such as searching, browsing, or having a robot guess "People You May Know," you can find new friends and add them to your list. Having other users as friends grants you certain privileges to their information. You may have access to their Profiles even if they're set to "Private"; you see their updates on your feed; and, most important to us, friends can interact inside apps.

That's MySpace in a nutshell. You sign up, bling out your Profile, add some friends, upload some pictures, install some apps, and send some messages.

What Is OpenSocial?

OpenSocial is a specification that defines how Web sites allow third-party apps to run on their sites. That means many things. For example, OpenSocial defines what information a Web site can and must expose for each of its users. It defines a client-side JavaScript runtime environment with a set of APIs that describe how to fetch and update that user data. This environment is commonly called an "OpenSocial container." Likewise, it defines a set of server-side APIs that manipulate the same data but can be called from a server running your language of choice, such as PHP. It also defines a markup language called OSML that can be used to draw your app and fetch data.

This specification was started by the folks at Google, along with the help of MySpace, Hi5, and other initial early adopters. Maintenance of the spec has since been passed to the OpenSocial Foundation and the community. That means any implementers, such as MySpace, Orkut, and LinkedIn, app developers, or really anyone at all can suggest modifications to the spec and participate in setting the direction. Those modifications are then voted on in a public forum and either brought into the spec (found at http://opensocial.org) or rejected.

Any site on the Web is free to implement the OpenSocial spec. The idea is that when a site becomes OpenSocial-compliant, it is able to run almost any of the thousands of apps already created using OpenSocial. If a site has correctly implemented all the various APIs, an app running on MySpace should run just as well anywhere else. That's the theory, anyway. In practice there are differences, some small, some large, between the various OpenSocial implementers, which can make it tricky to maintain an app on multiple sites.

What Is the MySpace Open Platform?

The MySpace Open Platform (sometimes called MySpace Developer Platform or MDP) is simply MySpace's implementation of the OpenSocial spec. MDP is a blanket term that covers how to sign up as a developer, how to update your app, and how to actually run your app.

MySpace tries its best (we really do) to fully implement the OpenSocial spec, but there are bound to be some inconsistencies. Most of these inconsistencies are small, but we'll cover them as they come up throughout this book. MySpace has also implemented some features that aren't in the spec; we'll cover these as well.

More than on any other OpenSocial-enabled site, developing apps on the MySpace platform gives you access to a huge number of users and potential customers. As of this writing, MySpace has more than 130 million monthly active users and is growing every day. Forty-five million of those users have installed apps. To give you an idea of the demographics, 53% of app users are female, with an average age of 18 to 24. That is a very marketable demographic to have at your fingertips.

What Tools Will You Need?

Here's where we finally start to get a little technical. After many long nights spent debugging why that callback wasn't being fired or where that stupid 401 unauthorized error was coming from, we've found some really useful tools that will make your life easier. Let's repeat that for effect: These tools will make your life easier.

Browsers

First, a quick note on browsers. Install Firefox right now. Go to www.firefox.com. In our opinion it's best to develop on Firefox, get your app running the way you like it, then worry about running it in Internet Explorer (IE) as well. Developing on the Web is still an "IE and everyone else" process. You get an app working in Firefox, and that means it will probably work just fine in Safari, Chrome, and Opera. Then you can bang your head for a few hours (days?) fighting floating divs and onclick handlers in IE.

Firebug

Firebug is the main reason Firefox is the Web developer's browser of choice. Firebug is an incredible tool for all Web developers, and every single one of us should install this

Firefox extension. It's even more useful because all JavaScript "alert" and "confirm" functions are blocked by the OpenSocial container, so it really is the best way to debug. Get it at http://getfirebug.com.

Firebug has many useful functions. We won't go into every last one here, but there are a few that we use every single day. To follow along at home, install the Firebug extension, restart Firefox, navigate to your Web page of choice, then click Tools → Firebug → Open Firebug. The Firebug window will open at the bottom of the browser.

Inspect

The Inspect feature allows you to view and edit CSS and inline style on any HTML element currently on the page. Click Inspect at the top left of the Firebug window and hover your cursor over a Web page. You'll see that the various HTML elements become highlighted. Clicking on one will show you the HTML markup of that element in the left pane and its current style in the right pane. You can then add a style or modify the existing style on the fly.

This is very useful for building up user interface (UI) elements quickly. Instead of making a change, saving a file, uploading it, and reloading the page in a browser, you can just edit the style very quickly and try different values. When you find what works in Firebug, you can make one edit to your page and it should be good to go.

Console

The Console tab shows you a couple of useful things. The first useful bit of information is that any and all JavaScript errors are displayed here, with a description of the error along with its file name and line number. The second is that any outgoing Web service requests are shown here. You'll be able to view the exact outgoing URL, any parameters and headers sent along, and the response from the server. As most MySpace requests are sent as outgoing Web service requests, this is an invaluable debugging tool. You'll be able to see if the correct parameters were appended to the request, and you won't have to output the response onto a div on the UI to see if you got the correct response or some sort of error.

Script

The Script tab is your debugger. You can load any HTML or JavaScript file into the left pane by selecting the file in the drop-down above that pane. You can then set breakpoints at any line with a green line number. In the right pane you can set watches on variables, view the current stack, and manipulate the breakpoints you've set.

With the Watch window you can interrogate any object by simply typing the variable name into the New Watch Expression bar. Doing so displays the object's current state and value. But you can also directly manipulate objects. Say you wanted to see what would happen if a Boolean variable were `true` instead of `false`; you can set that directly by entering something like `varName = true` into the Watch Expression bar.

Need to figure out what's causing that JavaScript error? Set a breakpoint on that line and inspect the objects and variables that are causing the issue. Need to figure out exactly what the response looks like from a request to the MySpace servers? Set a

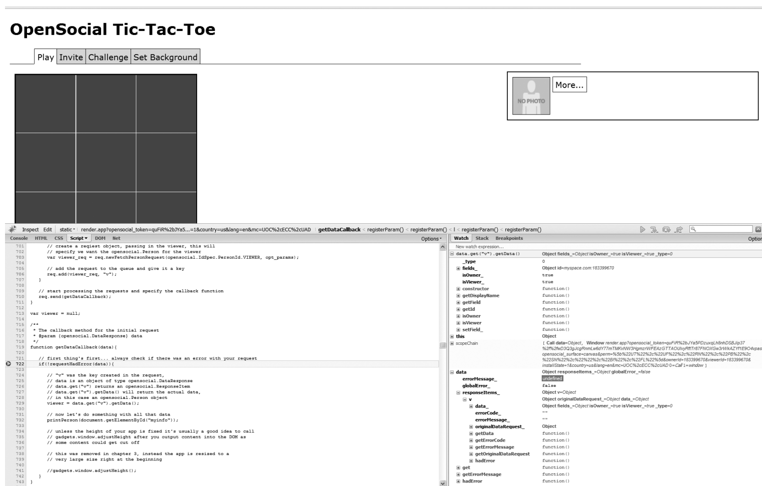

Figure I.2 Interrogating the response from MySpace's servers using Firebug.

breakpoint in your callback function and add the response object to your Watch list (see this in action in Figure I.2).

There are countless other things you can do with the Script tab, and Firebug in general, but we can't discuss them all here. We hope we've sold you on its usefulness and that you'll discover the many great features it has to offer.

Fiddler

Fiddler is a free tool that watches all the traffic coming into and out of your computer. With respect to building apps, it is useful mainly if you decide to use the server APIs as opposed to the JavaScript APIs. We'll discuss what that means later on in the book, but keep this tool in mind when you get there. To give you a little bit of a preview, you'll be sending requests for data to MySpace, and those requests need to be sent using an authentication scheme. Using the authentication scheme is tricky, but Fiddler will help you see exactly what you're sending to MySpace and exactly what MySpace is sending back to you. Get it here: www.fiddler2.com.

One other thing to note is that Fiddler works natively in Internet Explorer but not Firefox. If you do a little bit of Googling, you can get it up and running in both browsers.

2

Getting Basic MySpace Data

In this chapter we'll start with a very basic version of the Tic-Tac-Toe application and then expand on it to include examples of how you might fetch a user's name, address, and gender. We'll then show you how to use that information in your application. On the way, we'll cover how MySpace and OpenSocial define an `opensocial.Person` object and how to use that information to enrich your application.

We'll also get to really play with our Tic-Tac-Toe game. This will be our example app, and over the course of the book we'll take you from a basic game to a fully functional and feature-rich application.

At each step along the way we'll teach you new methods and programming tricks for accessing and using the OpenSocial API. Code examples will be shown in full and then broken down into segments for further explanation, allowing you to zero in on a specific feature or trouble spot.

The Two Concepts That Every Developer Should Know

Before we begin, there are two basic concepts that every MySpace and OpenSocial app developer should know about:

- In the world of MySpace apps, everyone is either an Owner or a Viewer.
- The MySpace permission model is a complex and confusing beast that will likely leave you frustrated and angry.

On that note, let's get started!

Basic Concepts: Owner and Viewer

The concept of Owner and Viewer is central to OpenSocial, and it boils down to

Owner = the MySpace user on whose surface the application is running

Viewer = the MySpace user who is currently interacting with the application

Once you start accessing MySpace data, all the requests you make are made in the context of the Owner and the Viewer.

For example, let's say John has the app installed and Jane is viewing John's MySpace Profile. John is the Owner and Jane is the Viewer. Alternatively, if John is looking at the app on his own MySpace Profile, he is both the Owner and the Viewer.

When you request data, you'll be asking for information about either the Owner or the Viewer.

Basic Concepts: Permissions for Accessing MySpace Data

Not all user data is accessible to apps, and the data that is accessible is subject to a sometimes convoluted permission model. The key idea behind the MySpace permission model (dubbed Open Canvas) is that under certain circumstances an app can access a user's data even if that user hasn't installed the app. However, this access can happen only on the app's Canvas surface and not on the Home or Profile surfaces.

Under this model there are essentially three core questions used to determine whether an app can access a piece of data:

1. **Does the Viewer have the app installed?**
2. **Is the Viewer logged in?**
3. **Has the Viewer blocked the app?** Blocking an app prevents any and all access to the Viewer's data.

Figure 2.1 is a quick reference chart for Open Canvas; remember, though, that these permissions can change. So stay on your toes and watch out for permission traps when accessing user data.

Starting Our Tic-Tac-Toe App

The first thing we need to do is to get the basic Tic-Tac-Toe app started. Since this app is just scaffolding on which to hang OpenSocial concepts, we're going to gloss over a lot of the details. At this initial level our app could just as easily be a JavaScript game embedded in a Web page. It is the OpenSocial code that we will add later that will make the game come alive.

> **Note**
>
> Before we start, you should have the basic version of Tic-Tac-Toe installed as an app on your MySpace developer account. You will find the code on our Google Code page at http://code.google.com/p/opensocialtictactoe/source/browse/trunk/chapter2/chapter2ttt.html. For instructions on how to create, publish, and edit an app, please refer to Chapter 1, Your First MySpace App.

The game is a simple table grid of nine squares. Each square has a click handler to record player moves. Underneath the surface is a state-of-the-art computer AI (i.e., random-move engine) to play against.

	Viewer has added the app	Viewer hasn't added the app	Viewer is logged out	Viewer has blocked the app
Basic Owner data* (Home)	Yes	Yes	Yes	Yes
Basic Owner data* (Profile)	Yes	Yes	Yes	Yes
Basic Owner data* (Canvas)	Yes	Yes	Yes	Yes
Basic Viewer data* (Home)	Yes	No	No	No
Basic Viewer data* (Profile)	Yes	No	No	No
Basic Viewer data* (Canvas)	Yes	Yes	No	No
Owner's friend list	Yes	Yes	Yes	Yes
Viewer's friend list	Yes	No	No	No
Owner's public media**	Yes	Yes	Yes	Yes
Viewer's public media**	Yes	No	No	No

*Basic person data is defined as the user's display name, URL, Profile picture, and user ID.
** "Media" includes photos, albums, and videos.

Figure 2.1 MySpace Open Canvas permission model for accessing user data.

The remainder of this chapter will introduce some basic concepts that are central to OpenSocial and apply them to our Tic-Tac-Toe app. At the end of this chapter, our app will display the current player's name, gender, current location, and Profile image.

Accessing MySpace User Data

MySpace Profile data is represented in OpenSocial by the `opensocial.Person` object. Essentially,

`opensocial.Person` = a MySpace Profile

The `opensocial.Person` object contains the methods shown in Table 2.1.

In addition, the `opensocial.Person` object supports a wide range of data fields, such as a user's ID, Profile picture, or favorite movies. Table 2.2 presents all of the supported person fields along with their return types.

Table 2.1 **opensocial.Person Methods***

Method	Purpose
`getDisplayName()`	Gets a text display name for this person; guaranteed to return a useful string
`getField(key, opt_params)`	Gets data for this person that is associated with the specified key
`getId()`	Gets an ID that can be permanently associated with this person
`isOwner()`	Returns `true` if this person object represents the owner of the current page
`isViewer()`	Returns `true` if this person object represents the currently logged-in user

*Portions of this chart are modifications based on work created and shared by Google (http://code.google.com/apis/opensocial/docs/0.8/reference/#opensocial.Person) and used according to terms described in the Creative Commons 2.5 Attribution License (http://creativecommons.org/licenses/by/2.5/).

Table 2.2 **MySpace opensocial.Person Fields***

MySpace-Supported Person Field	Description	Return Type
`opensocial.Person.Field.ABOUT_ME`	A general statement about the person	String
`opensocial.Person.Field.AGE`	Person's age	Number
`opensocial.Person.Field.BODY_TYPE`	An object containing the person's body type; MySpace returns only `opensocial.BodyType.Field.BUILD` and `opensocial.BodyType.Field.HEIGHT`	`opensocial.BodyType`
`opensocial.Person.Field.BOOKS**`	Person's favorite books	Array of strings
`opensocial.Person.Field.CHILDREN`	Description of the person's children	String
`opensocial.Person.Field.CURRENT_LOCATION`	Person's current location; MySpace returns only `opensocial.Address.Field.REGION`, `opensocial.Address.Field.COUNTRY`, and `opensocial.Address.Field.POSTAL_CODE`	`opensocial.Address`
`opensocial.Person.Field.DATE_OF_BIRTH`	Person's date of birth	Date
`opensocial.Person.Field.DRINKER`	Person's drinking status (with the enum's key referencing `opensocial.Enum.Drinker`)	`opensocial.Enum`

Table 2.2 ***Continued***

MySpace-Supported Person Field	Description	Return Type
opensocial.Person.Field.ETHNICITY	Person's ethnicity	String
opensocial.Person.Field.GENDER	Person's gender (with the enum's key referencing opensocial.Enum.Gender)	opensocial.Enum
opensocial.Person.Field.HAS_APP	Whether or not the person has added the app	Boolean
opensocial.Person.Field.HEROES**	Person's favorite heroes	Array of strings
opensocial.Person.Field.ID	MySpace user ID	String
opensocial.Person.Field.INTERESTS**	Person's interests, hobbies, or passions	Array of strings
opensocial.Person.Field.JOBS	Jobs the person has held; MySpace returns only opensocial.Organization.Field.NAME and opensocial.Organization.Field.TITLE	Array of opensocial.Organization
opensocial.Person.Field.LOOKING_FOR	Person's statement about whom or what he or she is looking for or what he or she is interested in meeting people for	opensocial.Enum
opensocial.Person.Field.MOVIES**	Person's favorite movies	Array of strings
opensocial.Person.Field.MUSIC**	Person's favorite music	Array of strings
opensocial.Person.Field.NAME	An object containing the person's name; MySpace returns only opensocial.Name.Field.UNSTRUCTURED	opensocial.Name
opensocial.Person.Field.NETWORK_PRESENCE	Person's current network status (with the enum's key referencing opensocial.Enum.Presence)	opensocial.Enum
opensocial.Person.Field.NICKNAME	Person's nickname	String
opensocial.Person.Field.PROFILE_SONG	Person's Profile song	opensocial.Url
opensocial.Person.Field.PROFILE_URL	Person's Profile URL	String (must be fully qualified)
opensocial.Person.Field.RELATIONSHIP_STATUS	Person's relationship status	String

Table 2.2 ***Continued***

MySpace-Supported Person Field	Description	Return Type
`opensocial.Person.Field.RELIGION`	Person's religion or religious views	String
`opensocial.Person.Field.SEXUAL_ORIENTATION`	Person's sexual orientation	String
`opensocial.Person.Field.SMOKER`	Person's smoking status (with the enum's key referencing `opensocial.Enum.Smoker`)	`opensocial.Enum`
`opensocial.Person.Field.STATUS`	Person's status	String
`opensocial.Person.Field.THUMBNAIL_URL`	Person's photo thumbnail URL	String (must be fully qualified)
`opensocial.Person.Field.TV_SHOWS`	Person's favorite TV shows	Array of strings
`opensocial.Person.Field.URLS`	URLs related to the person or the person's Web pages or feeds; returns only the Profile URL, so it's better to use `opensocial.Person.Field.PROFILE_URL` instead	Array of `opensocial.Url`
`MyOpenSpace.Person.Field.MEDIUM_IMAGE`	Medium-sized version of the image returned by `THUMBNAIL_URL`; MySpace returns `opensocial.Url.Field.ADDRESS` and `opensocial.Url.Field.TYPE`. This field is not part of the OpenSocial spec and is a MySpace-specific extension.	`opensocial.Url`
`MyOpenSpace.Person.Field.LARGE_IMAGE`	Large version of the image returned by `THUMBNAIL_URL`; MySpace returns `opensocial.Url.Field.ADDRESS` and `opensocial.Url.Field.TYPE`. This field is not part of the OpenSocial spec and is a MySpace-specific extension.	`opensocial.Url`

*Portions of this chart are modifications based on work created and shared by Google (http://code.google.com/apis/opensocial/docs/0.8/reference/#opensocial.Person.Field) and used according to terms described in the Creative Commons 2.5 Attribution License (http://creativecommons.org/licenses/by/2.5/).

**The data response for these fields is unstructured and is always an array of one element. For example, a user's heroes aren't separated out into array elements; instead, the response is a text blob. This is because these fields on a MySpace Profile are free text.

Accessing Profile Information Using the opensocial.Person Object

The first thing we're going to add to our basic Tic-Tac-Toe app is the ability to get the default fields for our Viewer by fetching the corresponding `opensocial.Person` object using an `opensocial.DataRequest`. MySpace defines these default `opensocial.Person.Field` values for the `Person` object:

- Profile URL, or `opensocial.Person.Field.PROFILE_URL`
- Image URL, or `opensocial.Person.Field.THUMBNAIL_URL`
- Display name, or `opensocial.Person.Field.NAME`
- User ID, or `opensocial.Person.Field.ID`

These are the bare-minimum default fields that, as long as you have access to the `Person` object, will always be returned.

One thing to note is that all requests to the MySpace API are asynchronous, meaning you launch them as "fire and forget" requests. Typically you make a request and specify a callback function; when the server is done processing your request, it passes the results as a parameter into your callback function. One result of this is that you're probably going to want to put your data requests at the start of the code. That way the MySpace API can start processing your request as soon as possible, and your app won't be stuck waiting for data to come back.

How Asynchronous JavaScript Requests Work

JavaScript is capable of making synchronous or asynchronous data requests back to servers on the current page's originating domain. Under the hood this is accomplished via the `XMLHttpRequest` object. This may be alternately referred to as XHR or Ajax.

OpenSocial wraps this functionality up in the `opensocial.DataRequest` object, so you don't need to know all the gory details. Because this is always done asynchronously with OpenSocial, your code needs to specify a callback function to handle the server response.

JavaScript in the browser is inherently single-threaded, which basically means that it can do only one thing at a time. Asynchronous processing allows your app to not "freeze" while it's waiting for API requests to the MySpace servers to respond. A callback function allows your app to resume processing when it gets the data response.

Let's take a look at a function that does the following:

1. Instantiates an `opensocial.DataRequest` object
2. Calls `newFetchPersonRequest` to create a generic request object, passing in the `VIEWER` as a parameter
3. Adds the request to the queue by calling `add()`, passing in a key to name the request

The following code shows this function:

```
/**
 * Makes the initial call for the Viewer's opensocial.Person
 * object and requests all fields supported by MySpace
 */
function getInitialData(){

   // Returns an opensocial.DataRequest object
   var req = opensocial.newDataRequest();

   // Create a request object, passing in the Viewer; this will
   // specify we want the opensocial.Person for the Viewer
   var viewer_req = req.newFetchPersonRequest(opensocial.IdSpec.PersonId.VIEWER);

   // Add the request to the queue and give it a key
   req.add(viewer_req, "v");

   // Start processing the requests and specify the callback function
   req.send(getDataCallback);
}
```

The first line of the function simply returns an `opensocial.DataRequest` object. The `DataRequest` object is designed to take care of all your requesting needs. While developing our app, we'll make heavy use of it.

`opensocial.IdSpec.PersonId.VIEWER` is one of many enums you'll encounter in OpenSocial. In JavaScript this actually resolves into a string, and on MySpace specifically it is equal to the string `VIEWER`. You can use the string and enum interchangeably, but we recommend using the enum for two reasons:

- If you mistype the enum, your mistake triggers a JavaScript error, but if you mistype the string, it isn't immediately obvious that something went wrong, so it is harder to debug.
- The values to which the enums resolve aren't yet clearly outlined in the OpenSocial spec, so different containers might resolve enums to different strings.

We use `opensocial.IdSpec.PersonId.VIEWER` here to state that we want to fetch the Viewer's data. Had we wanted to fetch the Owner's data, we would have used `opensocial.IdSpec.PersonId.OWNER`.

`newFetchPersonRequest` takes the ID of the user you want to request as a parameter (in this case the Viewer) and returns an object. That object is actually unnamed by OpenSocial, and the documentation simply refers to it as a "request." This object is never used directly beyond how it's used here—passed into `DataRequest`'s `add()` function.

Essentially, the `DataRequest` object creates "request" objects that are passed into its own `add` function. It may seem a little confusing, but remember—you'll never have to use that mysterious, unnamed "request" object directly, and all OpenSocial API calls follow this general pattern.

The add function takes that request and adds it to a queue that is stored in the DataRequest object. We also give the request a key and, in this case, the string "v". This key is used to identify the response of a specific request; we'll use this key later when we talk about how to parse responses.

The last line tells the container to start processing the request queue by calling send(). We pass the name of our callback function into send() as a parameter; when the server is done processing our request, it calls this function while passing in the response. You can see the flow of the OpenSocial data request and response pattern in Figure 2.2.

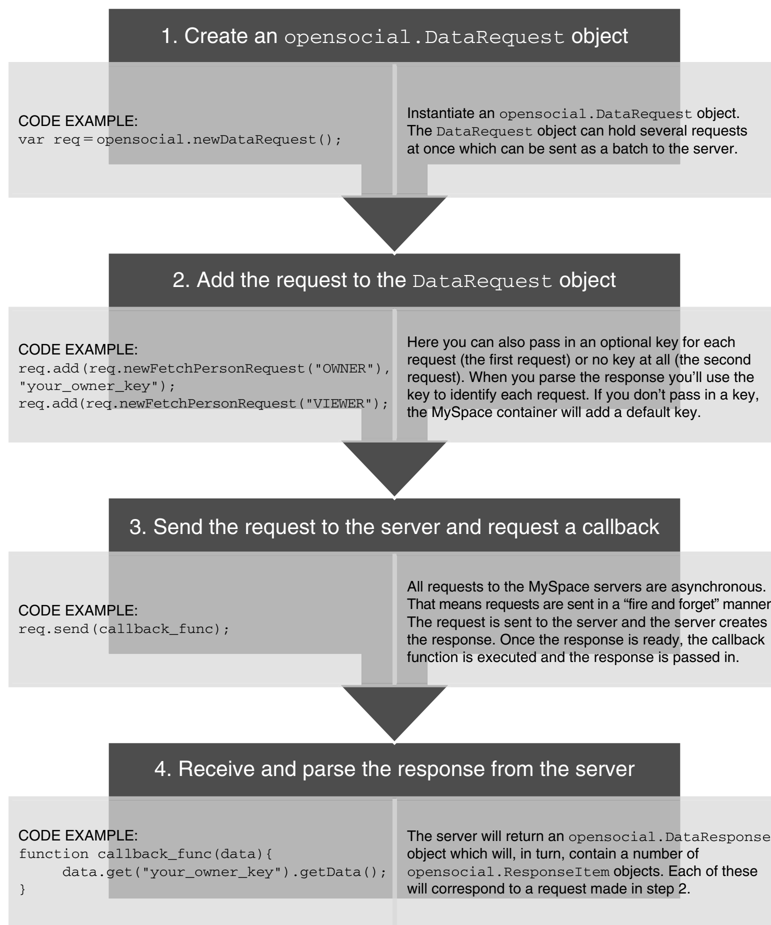

Figure 2.2 Basic request and response pattern for OpenSocial.

Getting More than Just the Default Profile Data

Fields beyond the default person fields typically follow a slightly different request pattern and may also require higher permission levels for access.

As an example of this, we'll be requesting our Viewer's status message (like a headline or shout-out), address, and gender. We've modified the function shown previously to now specify that we want this additional data:

```
/**
 * Makes the initial call for the Viewer's opensocial.Person
 * object and requests all fields supported by MySpace
 */
function getInitialData(){

   // Returns an opensocial.DataRequest object
   var req = opensocial.newDataRequest();

   // Used to specify what data is returned in the response
   var fields = [opensocial.Person.Field.CURRENT_LOCATION,
                 opensocial.Person.Field.GENDER,
                 opensocial.Person.Field.STATUS];

   // Create an empty object to use for passing in the parameters;
   // the parameters here are additional person fields from above
   var opt_params = {};

   // Add the list of fields to the parameters
opt_params[opensocial.DataRequest.PeopleRequestFields.PROFILE_DETAILS] = fields;

   // Create a request object, passing in the Viewer; this will
   // specify we want the opensocial.Person for the Viewer
   var viewer_req =
      req.newFetchPersonRequest(opensocial.IdSpec.PersonId.VIEWER,opt_params);

   // Add the request to the queue and give it a key
   req.add(viewer_req, "v");

   // Start processing the request and specify the callback function
   req.send(getDataCallback);
}
```

Let's take a look at what's changed. The first line is the same; we instantiate an `opensocial.DataRequest` object. In the next line we create an array of `opensocial.Person.Field` enums; this list corresponds to the specific `Person` fields we want from the API.

Next, an empty object is created called `opt_params`. This is another OpenSocial pattern that you'll see repeated in many places. Requests typically have some default action, and if you want to modify the default action, you need to supply some parameters to

define what you want to do. You do this by adding certain properties to an object and passing that object into the request.

When the MySpace container is processing a request for Viewer or Owner data, it checks the parameters sent in for a property named `profileDetail`. If it finds such a property, it knows that additional fields were requested. So let's take a look at that line again:

```
// Add the list of fields to the parameters
opt_params[opensocial.DataRequest.PeopleRequestFields.PROFILE_DETAILS] = fields;
```

On MySpace, `opensocial.DataRequest.PeopleRequestFields.PROFILE_DETAILS` resolves to the string `profileDetail`. We're creating that property and setting it equal to the array of `opensocial.Person.Field` values we created earlier. The container now knows that additional fields were requested.

The rest of the function is the same as before—we create the request, add it to the queue, and send it off.

opensocial.DataResponse and opensocial.ResponseItem (aka, Using MySpace User Data)

The various person fields are returned in different ways, but there are essentially three types of responses:

- A **literal**, such as a string or a number. For example, `opensocial.Person.Field.STATUS` returns a string.
- An **object**. For example, `opensocial.Person.Field.CURRENT_LOCATION` returns an `opensocial.Address` object.
- An **enum**. For example, `opensocial.Person.Field.GENDER` returns an `opensocial.Enum` enum object.

All responses from the API come in the form of an `opensocial.DataResponse` object. The `DataResponse` object contains some number of `opensocial.ResponseItem` objects; each `ResponseItem` corresponds to the requests we added to the `DataRequest` queue. For now, we just have the one request for the Viewer; we'll look at dealing with multiple requests and `ResponseItem` objects in the next chapter.

To get a particular `ResponseItem` object from an `opensocial.DataResponse`, you need to use a key. If you already provided an optional key, use that. Otherwise, you can use the default key created by the MySpace container.

> **Tip: Use an Optional Key**
>
> It's recommended that you always use an optional key, as the default keys can be tricky to use. But, if you're curious, the container-generated keys are found in the `MyOpenSpace.RequestType` namespace.

```
/**
 * The callback method for the initial request
 * @param {opensocial.DataResponse} data
 */
function getDataCallback(data){

   // "v" was the key created in the request,
   // data is an object of type opensocial.DataResponse
   // data.get("v") returns an opensocial.ResponseItem
   // data.get("v").getData() will return the actual data,
   // in this case an opensocial.Person object
   viewer = data.get("v").getData();

   // Now let's do something with all that data
   printPerson();
}
```

Let's take a look at this function more closely. The first thing to notice is that it is the function we specified when we called `send()` in our previous request function.

```
// Start processing the request and specify the callback function
req.send(getDataCallback);
```

The function `getDataCallback` is executed when the server has processed our request and prepared a response. The first line in `getDataCallback` takes that `DataResponse` object and calls the `get()` function, passing in our key. Above we named our Viewer request `"v"`; here we use that key to get the `ResponseItem` that corresponds to that request.

The function call `data.get("v")` returns the `ResponseItem` object that contains the Viewer's `opensocial.Person` object; we get the `Person` by calling `getData()`. Now the variable `viewer` contains an object of type `opensocial.Person`, which in turn contains the details of the Viewer, specifically the four default fields plus the current location, gender, and status. Table 2.3 shows the methods available for the `ResponseItem` object.

Table 2.3 **opensocial.ResponseItem Methods***

Method	Purpose
`getData()`	Gets the response data
`getErrorCode()`	If an error was generated by the request, returns the error code
`getErrorMessage()`	If an error was generated by the request, returns the error message
`getOriginalDataRequest()`	Returns the original data request
`hadError()`	Returns `true` if there was an error in fetching this data from the server

*Portions of this chart are modifications based on work created and shared by Google (http://code.google.com/apis/opensocial/docs/0.8/reference/#opensocial.ResponseItem) and used according to terms described in the Creative Commons 2.5 Attribution License (http://creativecommons.org/licenses/by/2.5/).

The next section of our code calls the function `printPerson()`, which is a helper function that we use to parse out the details of our Viewer and output them to the screen. Here we will parse out all three types of responses—enums, strings, and objects:

```
/**
 * Output the Viewer data onto the surface
 */
function printPerson(){
   if(null !== viewer){
      // You can set the src attribute of an
      // <img> tag directly with THUMBNAIL_URL

      document.getElementById("profile_image").src =
      ➥viewer.getField(opensocial.Person.Field.THUMBNAIL_URL);

      // getDisplayName is a shortcut for
      // getField(opensocial.Person.Field.NICKNAME)
      document.getElementById("name").innerHTML = viewer.getDisplayName();

      // Get the Viewer's status
      var status = viewer.getField(opensocial.Person.Field.STATUS);
      if(status && status.length > 0){

         // If the status has been set, append it after the name
         document.getElementById("name").innerHTML += " \"" + status + "\"";
      }

      // Get the opensocial.Address object
      var location = viewer.getField(opensocial.Person.Field.CURRENT_LOCATION);

      // The Address object is used similarly to the Person object; both pass a
      // field into a getField function
      document.getElementById("location").innerHTML =
      ➥location.getField(opensocial.Address.Field.REGION);

      // gender is an opensocial.Enum object
      var gender = viewer.getField(opensocial.Person.Field.GENDER);

      // getDisplayValue is defined by the specific container and can and
      // will differ between containers and is designed for displaying only
      document.getElementById("gender").innerHTML = gender.getDisplayValue();

      // The response of getKey is defined by OpenSocial and therefore
      // you can compare the result to some known value
      if(gender.getKey() === opensocial.Enum.Gender.FEMALE){
         document.getElementById("myinfo").style.backgroundColor =
            "#fcf"; // pink
      }
```

```
        else{
            document.getElementById("myinfo").style.backgroundColor = "#09f"; // blue
        }
    }
}
```

Let's break down this code. To access the data from the `viewer` variable, which is of type `opensocial.Person`, we'll typically need to use the `getField()` function. After we confirm that `viewer` isn't null, we call `getField()`, pass in the field we want as a parameter, and set that value to the `src` attribute of an image tag. The field we asked for, `opensocial.Person.Field.THUMBNAIL_URL`, is returned as a string, so we can just access it directly. That's really all there is to parsing out a `Person` field that is returned as a string. For details on the return type for every supported `Person` field, refer back to Table 2.2 in this chapter.

The next line uses a shortcut to retrieve the Viewer's display name. Another shortcut, `getID()`, exists to retrieve the Viewer's ID. All other fields are accessed using `getField()`, as you'll see in the next few lines. We could have used `getField (opensocial.Person.Field.NICKNAME)` to retrieve the display name, but the shortcut is cleaner.

Next, we parse out the Viewer's current location; this field is returned as an `opensocial.Address` object. You'll notice that the `opensocial.Address` object behaves similarly to the `opensocial.Person` object. Both have fields that describe the types of data you can retrieve, which in turn are fetched using the `getField()` function. Many of the objects in OpenSocial follow this pattern.

Finally, we parse out the gender, which is an `opensocial.Enum` object. There are two useful types of data in each `opensocial.Enum`: the display value and the key. The display value is retrieved using `getDisplayValue()` and is a string that is defined by each container as it sees fit. For example, the display value for the female gender, depending on the container, could be "female" or "It's a girl!" or even "http://example.com/girl.jpg." Because these display values can be different from container to container, the key is typically used when making comparisons as the key values are defined in the OpenSocial spec.

In our code, we output the display value to the surface but use the key to make a logical comparison; we set the background color to pink if the gender is female (i.e., it's equal to `opensocial.Enum.Gender.FEMALE`), or blue otherwise.

In the full code listing for the chapter, which you can find at http://code.google.com/p/opensocialtictactoe/source/browse/#svn/trunk/chapter2, we added a function that goes through each `Person` field and parses it. If you're wondering how to parse a particular field that wasn't covered here, take a look at the code and you should be able to find what you need.

Talking to Your Parent

The MySpace site itself runs on the domain myspace.com. All MySpace apps live on a domain different from that, usually msappspace.com, unless it's an external iframe app (see Chapter 9, External Iframe Aps, for details on those). Because of this difference in domains, the iframe in which your app runs can't access the outer page. The app is said to be in a "jail domain." Cross-domain access is a security measure built into all modern browsers, and MySpace makes use of it to prevent any malicious attacks on the myspace.com domain.

However, there are a few functions that are exposed to apps that do talk to the parent page. A cross-domain scripting trick called inter-frame procedure call (IFPC) opens a few holes in the jail domain for apps to take advantage of. Let's take a look at two of the more useful functions.

The first is `gadgets.window.adjustHeight(new_height)`. This function asks the parent page to resize the iframe vertically.

`new_height` can take on several types of values. It can be an integer greater than 1; in this case the iframe's height is set to the specified number in pixels.

It can be a fraction greater than 0 and less than or equal to 1. In this case the iframe is resized to try to fill the specified fraction of the page. A `new_height` of 0.5 would cause the iframe to fill half the available space, for example.

Last, `new_height` can be left out completely; in this case the iframe is resized to fill its contents.

You'll see in the app that we use `adjustHeight` like this:

```
gadgets.window.adjustHeight();
```

This causes the iframe to be resized to fill its contents. This is usually the best way to go, as it can be hard to know exactly how big your content is at all times. The other option is to resize your app to a very large number when it loads, such as 5000 pixels. This works well on the Canvas page but not on Home or Profile.

The other useful function is `gadgets.views.requestNavigateTo(view)`. This function causes the browser to navigate away from the current page to the specified view. It's how you take the user from the Home page to your Canvas page, for example. The supported view names are `gadgets.views.ViewType.HOME`, `gadgets.views.ViewType.PROFILE`, and `gadgets.views.ViewType.CANVAS`.

Let's take a look at a handy wrapper function:

```
// This function wraps gadgets.views.requestNavigateTo,
// view comes from the enum gadgets.views.ViewType
function rNT(view){

    // Get the list of views that the container currently supports;
    // This returns a hash table keyed by view name
    var supported = gadgets.views.getSupportedViews();
```

```
    // If the view name passed in is in the supported hash table
    if(supported[view]){

        // Request to navigate to that view
gadgets.views.requestNavigateTo(supported[view]);
    }
}
```

This function takes in the name of the view to navigate to and then gets the list of currently supported views by calling `gadgets.views.getSupportedViews()`. This function returns a hash table of `gadgets.views.View` objects, keyed by the names of the views.

If the list of supported views contains the specified view, `requestNavigateTo` is invoked and the `View` object is passed in. To make use of the `rNT` function, you might do something like this in the onclick handler of a button:

```
rNT(gadgets.views.ViewType.CANVAS);
```

That will cause the browser to navigate to the app's Canvas page.

Error Handling

JavaScript errors from apps seem to be an all-too-common occurrence, but they don't have to be. Most developers don't expect errors to occur, and so they don't test or prepare for them as a result, but errors do happen. Your best defense against them is to admit that yes, they do occur, and yes, you need to be ready for them.

OpenSocial offers a few functions you can use to deal with errors. In the following example we check for errors before we parse the response for data. If an error is found, the response data isn't parsed.

```
/**
 * Check if the response had an error; if it did, log it and if
 * it was an INTERNAL_ERROR attempt to retry the request
 * @param {opensocial.DataResponse} data
 */
function requestHadError(data){

   // Return true if data is null or undefined
   if(!data) return true;

   // Check the opensocial.DataResponse for the global error flag
   if(data.hadError()){
      // Find the specific opensocial.ResponseItem that had the error
      var ri = data.get(Tic-Tac-Toe.RequestKeys.VIEWER);
```

```
        if(ri && ri.hadError()){

            // Output the error message
            log(ri.getErrorMessage());

            // Check the error code; an INTERNAL_ERROR can simply mean
            // network congestion or MySpace server instability

            if(opensocial.ResponseItem.Error.INTERNAL_ERROR === ri.getErrorCode()){

                // Retry the request a certain number of times; make
                // sure you don't create an infinite loop here!

                if(retries > 0){
                    retries--;
                    window.setTimeout(getInitialData, 1000);
                }
            }
        }

        return true;
    }

    return false;
}
```

First, we check if the response is null or undefined, and then check if the global error flag was set in the `DataResponse` object by calling the object's `hadError()` function. This flag is set if any of the requests had an error. Each `ResponseItem` has an error flag as well, so we'll need to check each `ResponseItem` in order to find out which request had the error, if the global flag was set. We do this by calling the `ResponseItem`'s `hadError()` function. In this case we had only one request, so there is only one `ResponseItem`, but in the event of multiple `ResponseItem` objects we would check each one in turn to determine its error state.

Similar to an `opensocial.Enum` object that has two types of data, one for display and one for logical comparisons, each `opensocial.ResponseItem` that encounters an error also has two types of data. The error message, accessed by `getErrorMessage()`, is an arbitrary string that should describe the error and help you debug what happened. The error code, accessed by `getErrorCode()`, matches up to one of the codes found in the `opensocial.ResponseItem.Error` enum. Common causes for each type of error code may be found in Table 2.4.

You'll notice that we also attempted to retry the request in the event of an `INTERNAL_ERROR`. This is because an `INTERNAL_ERROR` can sometimes occur because of network congestion or other temporary problems. Your request might succeed if you wait a second or so and try again.

Table 2.4 **Common Error Code Causes**

Common Error Codes	Causes
opensocial.ResponseItem.Error.BAD_REQUEST	The most common cause of this error is that some part of your request didn't make sense to the container. For example, you passed in a negative value that should be positive, you didn't supply a required field, or you passed in an invalid ID for a particular request.
opensocial.ResponseItem.Error.FORBIDDEN	This error is returned only when the server responds with a status code of 403 and is not commonly seen.
opensocial.ResponseItem.Error.INTERNAL_ERROR	This is a catchall error that typically occurs when something unknown happens on the MySpace side of the request. As such, it should be intermittent. It can also occur in `opensocial.requestPermission` (a topic we'll cover in the section on requesting permissions in Chapter 3) if a permission was requested but no new permission was granted, or if the server returned an unknown status code.
opensocial.ResponseItem.Error.NOT_IMPLEMENTED	This error is returned if you've requested some OpenSocial functionality that MySpace doesn't support. If you receive this error, either the entire function you're calling isn't available, or some parameter in a request isn't supported. An example of the latter would be that you requested an `opensocial.Person` field that isn't supported.
opensocial.ResponseItem.Error.UNAUTHORIZED	This error most commonly happens when you've requested data to which you don't have access. It may be possible to use `opensocial.requestPermission` to ask the user to grant your app access to the data. Again, see Chapter 3 for details.

Handling errors offers two benefits:

- There are no embarrassing JavaScript errors when you try to retrieve objects that don't exist.
- There's a possibility that you can recover from your error by retrying the request.

> **Warning**
>
> When retrying your request after an error, be careful you don't code yourself into an infinite loop! It's best to keep a retry counter as we do in the code shown previously.

Summary

In this chapter we covered the basic flow of an OpenSocial application, focusing on the process of accessing data on the API and OpenSocial's request/response pattern. We also defined an `opensocial.Person` object—what it is, how to get it, and what to do with it.

This request/response pattern is the foundation of any OpenSocial app. You will use it every time you request data from the server, so we hope you were paying attention.

> **Note**
>
> Code listings and/or code examples for this chapter and every other chapter can be found on our Google Code page under http://opensocialtictactoe.googlecode.com.

Index

Symbols and Numbers

${sender} reserved variable, 81–82

0.7 container, 208–211

A

About section, Profile page, 276–278

access points, application security, 302

Acknowledge button, 267

activities

creating from app's Canvas surface, 79

get_activities_atom, 194–195

get_friends_activities_atom, 195

getting app listed on Friend Updates. *See* opensocial.requestCreate Activity

os:ActivitiesRequest tag, Data Pipeline, 223–224

Add App button, 69, 308

add function, 17

Add This App button, 7, 276

adjustheight function, cross-domain access, 23

AdSense, 311–313

advertisements

BuddyPoke app and, 320

creating with Cubics, 313–314

creating with Google AdSense, 311–313

creating with RockYou! ads, 314–316

monetizing Flixster Movies app through, 322

monetizing Playdom apps through, 326

TK's apps, 323

aggregation, of activity feeds, 82

Ajax (Asynchronous JavaScript and XML), 94, 200–203

albums

- creating with create_album, 195–196
- fetching, 41–42
- fetching with get_album, 184–185
- fetching with get_albums, 183

albums, porting to OpenSocial 0.9

- fetching, 333–335
- updating, 338–339
- using opensocial.Album, 330–333

ALL, not filtering friend list, 33

ALL_ALL, invariant message bundles, 256

Amazon

- cloud storage products, 64
- Web Service, 174, 298–299

app categories, 308

app data store, 47–56

- app data P2P play downsides, 147
- to avoid scaling bottlenecks, 298
- hacking, 302
- limitations of, 153
- overview of, 47–48
- refactoring to build local, 51–56
- saving and retrieving data, 48–51
- setting up AppDataPlay game, 127–133

App Denial and Status Clarification, MySpace Developer's Forum, 266–268

App Engine, Google, 64

app gallery, promoting app with, 306–308

app life cycle, 265–281

- changing app Profile/landing page, 275–278
- hiding and deleting app, 274
- making changes to live app, 274–275
- managing developers, 279–280
- publishing. *See* publishing app
- republishing live app, 275
- suspension and deletion of app, 280

app Profile, 275–278, 308

AppDataPlay game object, 125–133

Apple Dashboard widget format, 260

application security, 301–302

&appvers=dev, 275

app.yaml

- getting started with Google App Engine, 157–158
- sending messages using IFPC, 209
- server code for REST API, 181
- testing OAuth implementation locally, 166
- updating for friends Web service, 202

Aptana, and Unicode, 65

arrays, responseValues, 74

Asynchronous JavaScript and XML (Ajax), 94, 200–203

asynchronous JavaScript requests, 15–17

authentication. *See* OAuth

AUTHORIZATION parameter, gadgets.io.makeRequest, 95

automation candy, adding feed, 110–111

AWS (Amazon Web Service), 174

Azure, Microsoft, 174–175, 298–299

B

BAD_REQUEST error, 26

bandwidth, saving, 164

basic Profile, 204

batch requests, performance optimization, 292

blogs, 75–76

body items, 82–83

body, message, 75

Boku (Mobilcash), 317

Bootstrapper

- adding FriendPicker widget, 120–121
- initializing FriendPicker, 124–125

bottlenecks, identifying scaling, 295–297

branded pokes, 320

breakpoints, using Firebug as debugger, xxv–xxvi

broadband connection speeds, 285

browsers, developing MySpace with, xxiv

BuddyPoke app, 318–319

bulletins, 75–78

button text, customizing between views, 232–235

C

cache memory, scaling bottlenecks and, 295–296

callback function

- accessing more than just default Profile data, 18
- accessing Profile information, 15–17
- checking user permissions to access media, 44
- combating permission-denied errors in activities, 87–88
- defined, 15
- paging friend list, 200–202
- in requestSendMessage, 78–79
- in requestShareApp, 72–74
- setting up AppDataPlay game, 130–132
- using friends list data, 38–39
- using user data, 20

Canvas page

- accessing MySpace data on, 10
- creating activities from, 79
- creating "Hello World" app on, 4–6
- defined, 5
- MySpace messaging policies, 70
- permission model for accessing user data, 11
- sending notifications, 89
- spreading app to other users, 69

case sensitivity, language and country codes, 256

CDATA tags

- adding other surfaces to gadgets, 229–230
- fixing parsing errors in gadget code, 228–229
- JavaScript blocks using, 225

CDN (content delivery network), 92

ckeynsecret.py file, OAuth, 155

- clearAllGames method, 145
- clearGame method, 144–145

clearing, P2P game, 144–145

client-side code

- Data Pipeline tags processed in, 221–222
- implementing paging in, 200–203
- REST APIs, 197–199
- updating player bio lightbox, 236–237
- using custom templates in, 242–244

clients, and scaling performance, 298

cloud services

- Amazon Web Service, 174
- Google App Engine, 155
- overview of, 64
- pitfalls of deep linking, 93

codeveloper permissions, 279

comments

- adding to turn-based games, 135–138
- as supported message type, 74–76

communication
- external server. *See* external server communications
- viral features and. *See* viral features and communication

connection speeds, and performance, 285

Console tab, Firebug, xxv

constraints
- polling interval for real-time play, 146–147
- static content on external servers, 92

consumer key, MySpace
- defined, 153–154
- OAuth settings, 155–156
- server code for REST API, 183

container-generator keys, obtaining, 19

Content block, gadget XML
- adding surfaces to gadgets, 217–218, 229–230
- custom tag template definition, 240–242
- customizing button text between views, 232–235
- defined, 215
- getting information with os:ViewerRequest tag, 235–236
- merging Home and Profile with shared, 230–231
- reusing common content, 230–231
- shared style, 231–232
- subviews, 245–248
- using basic data, 218–219

content delivery network (CDN), 92

CONTENT_TYPE parameter, gadgets.io.makeRequest, 95, 97

control flow tags, OSML, 226–227

Cookie Jacker, 59–64

cookies, 56–64
- building Cookie Jacker app, 59–63
- external server security constraints for, 91
- overview of, 56
- reasons to not use, 57–59
- uses for, 64

country codes, 256

CPUs, scaling bottlenecks in databases and, 295–296

crackers, 302
- create_album function, 195–196

cross-app promotions, 323–325

cross-domain access, 23–24

cross-site scripting (XSS) attacks, 303

CSS styling
- editing app Profile page, 276–278
- responsive performance rules for, 285

Cubics ads, 313–314

culture codes, 255–256
- currentGame, clearing, 144–145

custom values, passing into app's Canvas surface, 86

D

data, getting additional. ***See*** **MySpace, getting additional data**

data, getting basic. ***See*** **MySpace, getting basic data**

data listeners, OSML, 250–254

Data Pipeline, 219–225
- coupling OSML with. *See* OSML (OpenSocial Markup Language)
- data tag os:ActivitiesRequest, 223–224
- data tag os:DataRequest, 223–224
- data tag os:PeopleRequest, 222–223
- data tags, 220–221
- data tags os:ViewerRequest and os:OwnerRequest, 222
- DataContext, 220
- defined, 214

Data Pipeline, *contd*.
displaying JSON results with data listener, 251–252
in-network vs. out-of-network data, 221–222
JavaScript blocks in OSML apps, 225
overview of, 219–220
working with, 235–237
data security, 301
data tags, Data Pipeline
in-network vs. out-of-network data, 221–222
os:ActivitiesRequest, 223–224
os:DataRequest, 223–224
os:PeopleRequest, 222–223
os:ViewerRequest and os:OwnerRequest, 222
overview of, 220–221
data types, creating activities, 80–81
data warehouses, for Internet-scale apps, 298
databases
data warehouses vs., 298
as primary scaling bottleneck, 295–297
DataContext, Data Pipelining, 220, 236–237
DataRequest object
accessing more than just default Profile data, 18–19
accessing Profile information, 15–17
app data store, saving and retrieving data, 48–51
asynchronous JavaScript requests and, 15
fetching albums, 333–335
fetching friend list, 31
fetching media, 39–43
fetching media items, 335–336
fetching Viewer's photos, 336–338
friend list filters and sorts, 32–33
os:DataRequest tag, Data Pipeline, 223–225
paging friend list, 32–37
updating albums and media items, 338–339
DataResponse object
error handling, 25
testing for errors, 300
using MySpace user data, 19–24
"Death Star" project, Enron, 284
debug flag, POST requests, 165
debugging, using Script tab of Firebug, xxv–xxvi
deep linking, 93
deleting app data, OpenSocial 0.8 and 0.9, 341–343
deleting apps, 274, 280
description, app, 307
design
adding feed reader to app, 98–104
turn-based games, 118–119
detailtype parameter, 203
Developer Addendum, 266
developers
managing, 279–281
signing up for MySpace account, 3–4
Developers & Testers option, My Apps page, 279–281
developers, interviews with successful, 318–326
Dan Yue (Playdom), 324–326
Dave Westwood (BuddyPoke app), 318–319
Eugene Park (Flixster Movies), 321–322
Tom Kincaid (TK's apps), 322–324
Development version, changing live app, 274–275
Dewoestine, Eric Van, 157

display content, gadgets, 215

display, designing feed reader, 103–104

display value, opensocial.Enum, 22

displayMode property, FriendPicker, 123

DOM (Document Object Model)
- adding feed reader to app, 93, 95
- customizing button text between views, 233
- handling raw XML content in, 98
- JSONP calls and, 112–113
- modifying script to use subviews and, 247–248
- processing client-side templates, issues with, 243
- processing RSS feed with FEED content type, 104, 106–107
- setting up feed reader, 101–102, 104
- TTT.List object references to, 37

domains, cross-domain access, 23–24

DRY (Don't Repeat Yourself) acronym, 230

duplicate applications, and app rejection, 271–272

dynamic content, creating, 92

E

e-mail accounts
- creating "Hello World" app, 4
- dealing with duplicate applications, 273
- signing up for MySpace developer account, 3–4

Edit App Information screen, 155, 216–217

Edit App Source, 274–275

Edit Profile, 276

endpoints
- Profile, 203–204
- supported by MySpace SDK. *See* REST API list
- testing for errors with OpenSocial, 300

Enron, 284

Enterprise-scale applications, 293

enums
- defined, 19
- parsing out, 21–22

error codes, common, 26

error flow, providing, 300

error handling
- for fault tolerance and stability, 300
- fixing parsing in gadget code, 228–229
- from makeRequest call, 100
- in on-site vs. off-site app, 200–202
- OpenSocial DataResponse objects, 300
- OpenSocial functions for, 24–27
- for performance optimization, 292

event handling, installs and uninstalls, 279–280

extended Profile, 204

external Iframe apps, 177–212
- cookie vulnerabilities not applicable to, 64
- pros and cons of, 177–178
- REST APIs. *See* REST (REpresentational State Transfer) APIs
- sending messages using IFPC, 208–212
- talking to parent page, 23

external server communications
- adding feed reader. *See* feed reader, adding to app
- adding image search, 111–114
- mashups, 92–93
- overview of, 91–92
- pitfalls of deep linking, 93
- posting data with form, 114

external servers
- defined, 91
- using Data Pipeline tags to pull in data from, 221
- using OAuth. *See* OAuth

F

Facebook, MySpace platform vs., 319, 321, 323, 325

failure array, responseValues, 74

"Fat Boy" project, Enron, 284

fault tolerance, 299–300

FEED content type, 100, 104–105

feed reader, adding to app, 93–111
- adding feed refresh option, 109–110
- feed automation candy, 110–111
- FEED content type, 104–105
- gadgets.io.makeRequest overview, 94–96
- handling JSON content, 97
- handling partial HTML content, 97
- handling RSS feed content, 97–98
- handling XML content, 98
- overview of, 93–94
- response structure, 96–97
- secure communication, 111
- setup and design of, 98–104
- TEXT content type, 107–108
- "user's pick" feed reader, 98
- XML content type with parsing, 105–107

feedback, for turn-based game play, 135–138

FeedBurner, 105

feedCallback function, 100

fetchFriendList( ) function
- fetching friend list, 31
- paging, 33–34
- sorting and filtering, 32
- using data, 37–39

fetchPhotosList function, 336–338

Fiddler, xxvi

fields
- accessing more than just default Profile data, 18–19
- MyOpenSpace.Album, 42
- MyOpenSpace.NotificationButton, 89
- MyOpenSpace.Photo, 40
- MyOpenSpace.Video, 43
- opensocial.Person, 11–17
- Profile endpoint, 203–204
- using MySpace user data, 19–24

filters
- app gallery, 306–308
- fetched lists, 31–33

finishing, P2P game, 144–145

Firebug, 48, xxiv–xxvi

Firefox, xxiv–xxv

Flixster Movies, 321–322

FORBIDDEN error, 26

form posts, communicating with external servers, 114

fr-CA (French Canada) culture code, 255

fr-FR (French global) culture code, 255

friend list
- calling requestShareApp, 72
- fetching, 30–31
- using data, 37–39
- using filters and sorts, 31–32
- using paging, 32–37
- Friend Updates, getting app listed on. *See* opensocial.requestCreate Activity

friendClickAction property, FriendPicker, 123

FriendPicker
- adding, 119–121
- operation modes, 122

FriendPicker, *contd*.
using in turn-based games, 121–125

friends
adding as developers, 279
displaying with repeater, 237–238
get_friends function, 185–187
get_friends_activities_atom function, 195
get_friendship function, 187–188
interacting with on MySpace, xxii–xxiii
prefetching record lists for paging, 287–291
Web service and paging, 200–203

friends object, 211–212

friendsCatalog property, FriendPicker, 123

friends_obj parameter, 198

friends.py script, 202–203

full Profile, 204

function signatures
defining requestSendMessage, 75
defining requestShareApp, 70–71
fetching albums, 42
fetching photos, 40–41
fetching videos, 42

G

gadget XML, 214–219
adding other surfaces, 229–230
adding second surface, 217–218
basic structure, 215
creating "Hello World", 214–217
creating initial file from existing code, 227–228
declaring and using basic data, 218–219
defined, 214
defining basic app meta-information, 216–217
including translations in app and testing, 259–260
internationalization and message bundles, 255–260
using UTF-8 encoding, 259

gadgets.io.makeRequest
application security and, 301
feed reader, adding to app, 93
feed reader, setting up and designing, 100, 102
feed refresh option, adding, 109–110
Google providing implementation code for, 105
making real MySpace requests, 170–173
making requests back to GAE server, 157
making signed POST request, 162–166
myspace:RenderRequest tag and, 226
option parameters to, 95–96
os:HttpRequest tag equivalent to, 221
overview of, 94–96
performance ramifications of, 286
requesting data with Data Pipeline tags using, 221
spicing up Home and Profile surfaces, 173–174

gadgets.log, 126

gadgets.util.escapeString, 108

gadgets.views.requestNavigateTo(view) function, 23–24, 245–248

GAE (Google App Engine)
Amazon Web Service vs., 174
getting started with, 157–158
making signed POST request using OAuth to, 162–166
making simple GET request to, 158–162
OAuth settings, 155–157

GAE (Google App Engine), *contd.*
supported data store properties, 158–159
testing OAuth implementation locally, 166–169
game engine, supporting P2P game play, 133–135
game play. *See* P2P (person-to-person) game play
GameInfo storage object, 125, 138–139
gameStatus function, 135
GET requests
making to GAE server, 158–162
real-world implications of, 164
testing OAuth implementation locally, 166–168
get_activities_atom function, 194–195
get_album function, 184–185
get_albums function, 183–184
getCurrentGameObject function, 138–139, 141
getData function, 20, 72–74
getDataCallback function, 20
getDisplayValue function, 22
getErrorCode function, 73
getErrorMessage function, 25
getField function, 22
getFriendGameData function, 147
get_friends function, 185–187
get_friends_activities_atom function, 195
get_friendship function, 187–188
getGameMovesString function, 143–144
getID function, 22
get_indicators function, 196
getInitialData function, 51
getMarkUp function, 36–37
get_mood function, 187–188
get_moods function, 188
get_photo function, 190
get_photos function, 188–190
get_profile function, 190–191
get_status function, 191
GET_SUMMARIES parameter, gadgets.io.makeRequest, 95, 97
Getter object, MySpace requests, 169–172
get_video function, 193
get_videos function, 192–193
global (invariant) culture, 255, 260
globally unique identifiers (GUIDs), 178
Gmail accounts, 273
Google
AdSense, 311–313
App Engine. *See* GAE (Google App Engine)
cloud storage products, 64
Gadgets specification, 260
gadgets.io.makeRequest code and, 105
Translate, 258
GQL (Graphical Query Language), 162
grid-computing, as storage solution, 64–65
GUIDs (globally unique identifiers), 178

H

hackers
avoiding game play, 166
security and, 302
hadError( ) function
checking user permissions to access media, 44
error handling, 25
requestShareApp callback, 73
hard disks, scaling bottlenecks in databases, 296
hardware
performance issues, 284
scaling bottlenecks in databases, 295–297

HAS-APP, 33

hasPermission function, 43–45

HEADERS parameter, gadgets.io.makeRequest, 95

"Hello World"
- creating app, 3–4
- entering app source code, 4–6
- for gadgets, 214–217
- installing and running app, 7
- signing up for developer account, 3–4

Hi5, importing apps to, 326

hiding apps, 274

high priority, defining in OpenSocial, 80

Home page
- creating shared style Content blocks, 231–232
- customizing button text between views, 232–235
- defined, 5, xxii–xxiii
- getting app listed on Friend Updates, see opensocial.requestCreateActivity
- indicating new notification on, 88
- merging with Profile page, 230–231
- not accessing MySpace data on, 10
- spicing up surface, 173–174
- spreading app to other users, 68–69

horizontal scaling, 297–298

href attribute, opensocial.requestShareApp, 85

HTML
- adding custom elements to About section of Profile page, 276–278
- adding feed reader to app, 97
- building feed reader UI, 99–101
- fragment rendering in OSML using, 248–250
- Inspect feature of Firebug using, xxv
- learning in order to use MySpace, 321
- widget formats using, 260

I

icon, app, 307

id string, 40, 42

IDs
- calling requestShareApp, 72
- defining requestSendMessage, 75
- sorting friend list by, 33

IdSpec object
- defined, 30–31
- fetching friend list, 31
- setting up AppDataPlay game, 131–132

IFPC (inter-frame procedure call)
- cross-domain access, 23
- sending messages, 208–212

Iframe apps. ***See*** **external Iframe apps**

iLike, app Profile page for, 276

image search, adding to app, 112–114

importing apps, to other social networks, 322

in-network data, Data Pipelining, 221–222

incentivized app installs, 310

indicators, get_indicators function, 196

infrastructure, and app data P2P play, 147

initializeGame( ) function, 51

inline tag templates, OSML, 239–244
- creating custom tag template definition, 240–242
- defined, 239
- using client-side, 242–244
- using custom tags, 242

inputs, validating, 299, 302–303

Inspect feature, Firebug, xxv

installing apps, event handling, 279–280

inter-frame procedure call (IFPC)
- cross-domain access, 23
- sending messages, 208–212

INTERNAL_ERROR, common cause of, 25–26

internationalization, and message bundles, 255–260
- creating first message bundle, 256–257
- creating translations of message bundle, 257–258
- culture code processing order, 255
- including translations in app and testing, 258–260
- limitations of message bundles, 260
- overview of, 255

Internet history, 250

Internet-scale applications, defined, 293

Internet-scale applications, performance guidelines
- data warehouses vs. relational databases, 298
- identifying scaling bottlenecks, 295–297
- knowing scaling point, 294–295
- load-testing system, 299
- overview of, 293–294
- pushing work out to nodes, 298
- remembering what you know, 297
- scaling horizontally, 297–298
- utility computing, 298–299

interviews with developers. *See* **developers, interviews with successful**

invariant (global) culture, 255, 260

Invite page
- client code for off-site app, 198–199
- creating link to, 204

Invite tab
- implementing, 180–182
- sending messages to users, 203–208
- updating to use OSML and Data Pipeline, 237–238

invite.py, 181, 202

isADraw function, 133–135

ISPs, database storage using, 64

J

jail domain, 23, 91

Java, hardware performance issues, 284

Java Virtual Machine (JVM), 284

JavaScript
- blocks in OSML apps, 225
- error handling in, 24–27
- information on using, 30
- learning in order to use MySpace, 321
- OSML vs., 219–220
- responsive performance rules for, 285
- sending messages using IFPC, 208–212
- TTT namespace, 30
- understanding asynchronous requests in, 15

JSLoader, 208–209

JSON (JavaScript Object Notation)
- adding feed reader to app, 97
- Ajax using, 94
- app data game store using, 125
- displaying results with data listener, 251–254
- error handling in off-site app, 201
- evaluating data in app data store, 48
- handling content for feed reader, 97
- makeRequest response object properties and, 96–97
- processing RSS feed with FEED content type, 104–105
- using simplejson file to manipulate strings, 160

JSP EL (JavaServer Pages Expression Language), 214

JVM (Java Virtual Machine), 284

K

keys
- MySpace secret and consumer, 153–154
- opensocial.Enum object, 22
- setting up AppDataPlay game, 127

Kincaid, Tom, 322–324

L

landing page, changing app, 275–276

latestGameInfo, 139

legal issues, deep linking, 93

libraries, OAuth, 154

lightbox, updating player bio, 236–237

Lightweight JS APIs, 330

literal data type, 81

literals
- defined, 19
- parsing out, 21–22

live app, making changes to, 274–275

Live version, making changes to, 274–275

load-testing, 299

loadAppData function, 52–53

loadAppDataCallback function, 130–131

loadFeed function, 100

loadFriendPicker function, 124

loadGame function, 139–142

loading issues, example of app rejection, 271–272

localization
- creating translations of message bundle, 257–258
- defined, 255

localRelay parameter, IFPC, 209

logging, debugging app, 126

logic flows
- designing turn-based games, 118
- for P2P game play, 138–144, 147
- setting up AppDataPlay game object, 125–133
- three-way handshakes as, 119

lookForWin function, P2P game play, 133–135

low priority, defining OpenSocial, 80

M

Mail Center, notification folder, 88

makePlayerMove function, 134, 135–138

makeRequest calls. *See* **gadgets.io.makeRequest**

marketing and monetizing, 305–327
- developer interviews. *See* developers, interviews with successful
- generating revenue with micropayments, 316–318
- overview of, 305
- promoting app on MySpace, 306–309
- user base and viral spreading. *See* viral spreading

markup
- OSML tags, 226
- simplifying for responsive performance, 285

mashups, 92–93, 98

MAX, paging friend list, 32–37

MDP (MySpace Developer Platform), xxiv

media items
- fetching albums, 41–42
- fetching photos, 39–41
- fetching videos, 42–43
- including on message template, 82–83
- using opensocial.requestShareApp, 86

media items, OpenSocial 0.9
fetching albums, 333–335
fetching in 0.8 and, 335–336
fetching media items, 335–336
fetching Viewer's photos, 336–338
opensocial.Album, 330–333
updating albums and media items, 338–339
uploading media items, 340–341
memory, scaling bottlenecks, 295–297
message bundles. *See* **internationalization, and message bundles**
message parameter, requestSendMessage, 75
messaging
MySpace policies for, 70
using requestSendMessage, 74–79
metadata, gadgets, 215
METHOD parameter, gadgets.io.makeRequest, 96
method stubs, for game play, 128–129
methods
opensocial.Person object, 11–14
opensocial.ResponseItem, 20
micropayments, generating revenue with, 316–318
Microsoft
Azure, 174–175, 298–299
cloud storage products, 64
Mobilcash (Boku), 317
Mobsters, 316, 325
modal dialogs, 69–70
ModulePrefs section, gadgets, 215, 216–217
modulo operations, 136
monetizing. *See* **marketing and monetizing**
mood
get_mood function, 187–188
get_moods function, 188
get_status function, 191
set_mood function, 195
multiple submissions, example of app rejection, 270–271
My Application screen, 265
MyAds, MySpace, 308–309
mylocalAppData, 52–53
MyOpenSpace.Album object, 41–42
MyOpenSpace.NotificationButton object, 89
MyOpenSpace.Photo object, 40
MyOpenSpace.requestCreateNotification, 88–90
MyOpenSpace.Video object, 42–43
MySpace
Agreement, 266
creating "Hello World" app, 3–6
Developer's Forum, 266
installing and running app, 7
Open Platform, xxiv
promoting app on, 306–309
Terms of Service, 310
Terms of Use. *See* Terms of Use
understanding, xxii
MySpace Developer Platform (MDP), xxiv
MySpace, getting additional data, 29–46
fetching albums, 41–42
fetching friend list, 30–31
fetching photos, 39–41
fetching videos, 42–43
using data, 37–39
using filters and sorts, 31–32
using paging, 32–37
MySpace, getting basic data, 9–27
accessing more than just default Profile data, 18–19
accessing Profile information, 15–17

MySpace, getting basic data, *contd*.
accessing user data, 11–14
error handling, 24–27
Owner and Viewer concepts, 9–10
permissions concepts, 10
starting Tic-Tac-Toe app, 10–11
using MySpace user data, 19–24
myspace:RenderRequest, 249–250

N

name, app, 307
names
reserved variable, 81–82
sorting friend list by nicknames, 33
namespaces
MyOpenSpace, 40–41
TTT. *See* TTT namespace
navigation
away from current page to specified view, 23–24
keystrokes used for, 122
to subviews, 245–248
using cookies for storage across surface, 64
using Pager object, 36
network distance, fetching friend list and, 31
New App Invite, 68–69
newFetchAlbumsRequest function, 333–335
newFetchMediaItemsRequest function, 335
newFetchPeopleRequest function
calling requestShareApp, 72
fetching friend list, 30–31
using data, 37–39
newFetchPersonRequest, 16
new_height function, for cross-domain access, 23
Next button, handling with paging, 36
NotificationButton object, 89
notifications
MySpace messaging policies, 70
sending, 88–90
send_notification function, 196–197
NOT_IMPLEMENTED error, cause of, 26
NUM_ENTRIES parameter, gadgets.io.makeRequest, 96

O

OAuth, 153–175
external server communication security with, 111
libraries, 154
MySpace incompatibilities with, 157
overview of, 153
secure phone home. *See* phoning home
setting up environment, 154–157
spicing up Home and Profile surfaces, 173–174
testing implementation locally, 166–169
understanding, 153–154
objects
defined, 19
parsing out, 21–22
off-site apps. *See* external Iframe apps
online references
Aptana, 259
basic, full and extended Profile data, 204
code examples for developing first app, 7
Cookie Jacker, 59
Cubics, 313–314
Fiddler, xxvi
Firebug, xxv
FriendPicker properties, 121

online references, *contd*.
GAE (Google App Engine), 157
Google AdSense, 311–313
micropayment companies, 316–318
MySpace SDK, 183
OAuth, 153
OpenSocial, xxiii
PayPal, 316
Python download, 155
RockYou! ads, 314–316
SDKs for accessing MySpace APIs, 154
Tic-Tac-Toe installation, 10
TortoiseSVN, 155
ONLINE_FRIENDS, filtering friend list, 33
Open Canvas, 10–11
OpenSocial
app data store. *See* app data store
basic request and response pattern for, 17
container, 40, xxiii
porting app to 0.9, porting app to OpenSocial 0.9
Sandbox tool, 214–217, 227–228
understanding, xxiii–xxiv
OpenSocial Markup Language. *See* OSML (OpenSocial Markup Language)
opensocial.Activity, 221
opensocial.Album, 330–333
opensocial.DataResponse. *See* DataResponse object
opensocial.hasPermission, 43–45
opensocial.IdSpec. *See* IdSpec object
opensocial.Message
defining requestSendMessage, 75–76
defining requestShareApp, 70–71
writing requestSendMessage code, 76–78
writing requestShareApp code, 71
opensocial.newUpdatePersonApp-DataRequest, 48
opensocial.Person
accessing more than just default Profile data, 18–19
accessing MySpace user data, 11–12
accessing Profile information, 15–17
Data Pipeline tags resulting in, 221
fetching friend list vs. fetching single, 30
request/response pattern, 19–24
opensocial.requestCreateActivity, 79–88
aggregation, 82
body and media items, 82–83
data types, 80–81
defining, 79–80
getting app listed on friend updates, 79–88
notifications patterned after, 88–89
overview of, 79
raising the event, 85–86
reserved variable names, 81–82
using activity callbacks to combat permission-denied errors, 86–87
using Template Editor to create templates, 83–94
using template system to create activities, 80
opensocial.requestCreateActivityPriority, 80
opensocial.requestPermission, 43–45, 87–88
opensocial.requestSendMessage, 74–79
callback for, 78–79
defining, 75–76
overview of, 74–75
writing code, 76–78
opensocial.requestShareApp, 67–74
callback for, 71–74
calling, 71

opensocial.requestShareApp, *contd*.
- communication policies and rules, 70
- defining, 70–71
- requestSendMessage signature vs., 75
- understanding, 67–69
- writing code, 71

opensocial.requestUploadMediaItem, 340–341

opensocial.ResponseItem
- error handling, 25–26
- requestShareApp callback, 72–74
- using MySpace user data, 19–24

opponent
- always considering computer as, 143
- creating custom tag template definition for, 240–242
- designing logic flow for turn-based games, 118–119
- implementing with P2P logic flow, 138–144

opponentPickedAction function, FriendPicker, 122–124

opt_callback parameter
- defining requestCreateActivity, 79
- defining requestSendMessage, 75
- defining requestShareApp, 71

optional keys, 19

opt_params object
- defined, 18–19
- defining requestShareApp, 71
- fetching albums, 42
- fetching photos, 40
- fetching videos, 42
- POST request, 172–173

Orkut, importing app to, 322

os-data section, Content block, 235–236

OS Lite, 330

os:ActivitiesRequest tag, Data Pipeline, 221–224

os:DataRequest tag, Data Pipeline, 221–224

os:Get tag, Data Pipeline, 248–249

os:HttpRequest tag, Data Pipeline, 221–222, 251–252

os:If control flow tag, OSML, 226–227

OSML (OpenSocial Markup Language)
- basic display tags, 226
- control flow tags, 226–227
- Data Pipeline, 219–225
- defined, 214
- gadget XML, 214–219
- overview of, 213–214
- remote content display tags, 226
- understanding, 225–226

OSML (OpenSocial Markup Language), advanced, 239–261
- data listeners, 250–254
- future directions, 260
- HTML fragment rendering, 248–250
- inline tag templates, 239–244
- internationalization and message bundles, 255–260
- working with subviews, 245–248

OSML (OpenSocial Markup Language), applying to Tic-Tac-Toe app, 226–238
- displaying data lists, 237–238
- reusing common content, 230–235
- setting up gadget, 227–230
- working with data, 235–237

os:OwnerRequest tag, Data Pipeline, 221–222

os:PeopleRequest tag, Data Pipeline
- defined, 221
- displaying friends list with repeater, 237–238
- overview of, 222–223
- processed on server, 221–222

os:ViewerRequest tag, Data Pipeline
defined, 221
getting Viewer information with, 235–236
overview of, 222
processed on server, 221–222
osx:Else control flow tag, OSML, 227
out-of-network data, Data Pipelining, 221–222
Own Your Friends app, 325
Owner
app data store. *See* app data store
concept of, 9–10
fetching app data for, 342
fetching friend list for, 30–31
getting ID of current, 169–170, 172
getting more than just default profile data, 19
os:OwnerRequest tag, 221–222
permission model for accessing user data, 10–11
reading app data from, 47–48
signed POST requests for authenticating, 162–163

P

P2P (person-to-person) game play, 117–149
adding user feedback, 135–138
advantages/disadvantages of, 148
finishing and clearing game, 144–145
fleshing out game logic, 138–144
"real-time" play, 146–148
supporting in game engine, 133–135
turn-based games. *See* turn-based gaming
pageSize property, FriendPicker, 123
paging
friend list, 199–203
prefetching record lists for performance optimization, 287–291
using, 32–37
parameters
custom tag template definitions, 241
gadgets.io.makeRequest, 95–96
Park, Eugene, 321–322
parsing
fixing errors in gadget code, 228–229
XML content type with, 105–106
Pay by Mobile button, Boku, 317
PayPal, 316–317
Pending status, publishing app, 266
performance, responsive
designing for, 284–285
designing for scale. *See* scaling performance
OpenSocial app guidelines, 285–292
overview of, 283
stability and fault tolerance, 299–300
understanding, 283–284
understanding scale, 284
user and application security, 300–303
permission-denied errors, combating, 87–88
permissions
accessing more than just default Profile data, 18–19
checking user settings, 43–45
codeveloper, 279
error code causes, 26
fetching photos uploaded to Profile, 39–41
"Hello World" app, 6
MySpace model for, 9–11
MySpace supported, 44–45
notifications, 90

persisting information (between sessions)
- setting up AppDataPlay game, 129–132
- using app data store. *See* app data store
- using cookies. *See* cookies
- using third-party database storage, 64–65

Person data type, 81

Person objects, see opensocial.Person object

person-to-person game play. ***See*** **P2P (person-to-person) game play**

phoning home, 157–173
- making real MySpace requests, 169–173
- overview of, 157
- signed POST request, 162–166
- testing OAuth implementation locally, 166–169
- unsigned GET request, 158–162

photos
- adding to Profile page, 276–278
- checking user permissions to access, 43–45
- fetching, 39–41
- fetching for Viewer in OpenSocial 0.9 and 0.8, 336–338
- fetching with get_photo function, 190
- fetching with get_photos function, 188–190

Play page, 198

Play tab
- porting to off-site app, 203–208
- requiring user's Profile data, 180

Playdom apps, 324–326

playerBioWrapper ID, 236–237

policies, MySpace communications or messaging, 70

pollForOpponentUpdatedMove function, 147

polling design, for "real time" play, 146–148

porting app to OpenSocial 0.9, 329–349
- fetching albums, 333–335
- fetching media items, 335–336
- fetching Viewer's photos, 336–338
- opensocial.Album, 330–333
- overview of, 329–330
- REST APIs, 343–348
- simplification of app data, 341–343
- updating albums and media items, 338–339
- uploading media items, 340–341

POST requests
- making real MySpace requests, 172–173
- making to GAE server using OAuth, 162–166
- real-world implications of, 164
- testing OAuth implementation locally, 166–168

POST_DATA parameter, gadgets.io.makeRequest, 96

Poster object, real MySpace requests, 169, 172–173

postTo function, 0.7, 210–211

power users, growing base of, 311

Prev button, 36

Preview Template button, Template Editor, 84

printPerson function
- getting information with os:Viewer-Request tag, 235–236
- updating player bio lightbox, 236–237
- using MySpace user data, 21

priorities
- defining in OpenSocial, 80
- defining opensocial.requestCreateActivity, 79

privacy, cookies and, 57

Profile page

accessing information using opensocial.Person, 15–17

accessing more than just default data, 18–19

accessing user data, permission model for, 11

bulletins, 77

changing, 275–276

comments on, 74

creating shared style Content blocks, 231–232

customizing button text between views, 232–235

defined, 5, xxii

defining for app, 4

defining requestShareApp on, 70–71

editing/updating user's, 75

endpoint, 203–204

fetching photos uploaded to, 39–41

get_profile function, 190–191

installing and running app, 7

linking to, 7

merging with Home page using shared Content blocks, 230–231

not accessing MySpace data on, 10

reserved variable names and, 81

spicing up surface using makeRequest, 173–174

as supported message type, 75–76

profileDetail property, 19

promoting app. ***See*** **marketing and monetizing**

properties

FriendPicker, 121–123

Google App Engine dat store, 158–159

makeRequest response object, 97

proto-mashups, 93

Prototype library, paging friend list, 200

proxy servers, makeRequest calls, 94

Publish button, 265–266

publishing app, 265–273

case study in successful rejection negotiation, 268–273

contesting rejection, 267–268

dealing with rejection, 267

overview of, 265–266

republishing live app, 275

why apps are not approved, 266

pushing work out to nodes, for Internet-scale apps, 298

Python editor, OAuth settings, 155

R

RAM memory, database performance, 296–297

readers-writers problem, solving, 51–53

"real-time" play, P2P games, 146–148

reason parameter, 70–71

recipients parameter, 70, 75

refactoring, to build local app data store, 51–56

refresh option, feed, 109–110

REFRESH_INTERVAL parameter, gadgets.io.makeRequest, 96, 109

registerOnLoadHandler directive, feed refresh, 109

rejection, app

contesting app, 267–268

dealing with app, 266

examples of, 270–273

reasons for app, 266

successful negotiation case study, 268–270

relational databases, data warehouses vs., 298

relay files, using IFPC, 209

remote content display tags, OSML, 226

removeGame method, 144–145

repeater, displaying friends list, 237–238

REpresentational State Transfer APIs. *See* **REST (REpresentational State Transfer) APIs**

republishing live app, 275

requestCreateActivity function. *See* **opensocial.requestCreateActivity**

requesting data, and performance, 286

requestPermission function, 43–45, 87–88

requestSendMessage function. *See* **opensocial.requestSendMessage**

requestShareApp function. *See* **opensocial.requestShareApp**

reserved variable names, 81–82

response structure, adding feed reader to app, 96–97

ResponseItem object. *See* **opensocial.ResponseItem**

responseJSON property, error handling when paging, 202

responseValues, arrays, 74

responsive performance. *See* **performance, responsive**

REST API list, 183–197
- create_album function, 195–196
- get_activities_atom function, 194–195
- get_album function, 184–185
- get_albums function, 183
- get_friends function, 185–187
- get_friends_activities_atom function, 195
- get_friendship function, 187–188
- get_indicators function, 196
- get_mood function, 187–188
- get_moods function, 188
- get_photo function, 190
- get_photos function, 188–190
- get_profile function, 190–191
- get_status function, 191
- get_video function, 193
- get_videos function, 191
- send_notification function, 196–197
- set_mood function, 195
- set_status function, 195
- test pages, 197

REST (REpresentational State Transfer) APIs, 178–208
- client code, 197–199
- friends response from, 211–212
- friends Web service and paging, 199–203
- how Web service is addressed, 178–179
- os:DataRequest tag resulting in endpoints, 221
- overview of, 178
- porting app to OpenSocial 0.9, 343–348
- Profile endpoint, 203–208
- server code, 181–183
- setting up external Iframe app, 179–180
- supported endpoints. *See* endpoints, supported by MySpace SDK
- supporting XML and JSON formats, 94

retrieving, friend's app data, 131–132

RockYou! ads, 314–316

RPC. *See* **IFPC (inter-frame procedure call)**

RSS feeds
- adding content to feed reader app, 97–98
- building feed reader UI, 98–101
- processing with FEED content type, 104–105

runOnLoad function, 198–199

S

Sandbox Editor
- declaring and using basic data, 218–219
- entering and executing code, 215–216
- using os:ViewerRequest tag, 236

Sandbox tool, 214–217, 227–228

Save Template button, Template Editor, 84

scaling
- advantages of app data P2P play, 147
- BuddyPoke app, 320
- Flixster Movies app, 322
- using external cloud servers for storage, 64–65

scaling performance
- application scale definitions, 293
- defined, 284
- Internet-scale app guidelines. *See* Internet-scale applications, performance guidelines

scaling point, 294–295

Script tab, Firebug, xxv–xxvi

scripts, 247–248

SDKs (software development kits)
- accessing MySpace APIs, 154, 183
- endpoints supported by MySpace, 183–197

secret key, MySpace, 153–154, 183

security
- app data store limitations, 153
- application, 301–302
- external server communications, 111
- fixing vulnerabilities, 311
- hacking and cracking, 302–303
- user data, 301

selectedFriend property, FriendPicker, 123

selectedOpponentDataCallback function
- app data game store, 131–132
- recognizing when new game has started, 145
- selecting opponent, 138–140

semaphore (flag value), readers-writers problem, 52–53

send( ) function, 17

Send Message, 74–76

send_notification function, 196–197

servers
- code for REST API, 181–183
- Data Pipeline tags processed on, 221–222
- external. *See* external server communications; external servers

Set Lots of Cookies option, Cookie Jacker, 63

setBackgroundClicked function, 43–44

set_mood function, 195

set_status function, 195

setTimeout directive, 52–53, 109–110

shared style Content blocks, 231–232

showCurrentPage method, 290–291

showFeedResults function, 103–104

showViewerBio function, 236–237

signatures
- function. *See* function signatures
- OAuth, 162–164

simplejson folder, 156–157, 160

Small/workgroup scale applications, 293

software development kits (SDKs)
- accessing MySpace APIs, 154, 183
- endpoints supported by MySpace, 183–197

sorting, fetched lists, 31–33

source code
- backing up before granting developer status, 279
- creating "Hello World" app, 4–6

stability, and responsive performance, 299–300

startGamePlayPolling function, 146–147

static content, and constraints, 92

status

get_status function, 191

set_status function, 195

storage

app data store. *See* app data store

cookies. *See* cookies

third-party database, 64–65

styles

custom tag, 241–242

shared style Content blocks, 231–232

Subversion, 155

subviews, OSML, 245–248

success array, responseValues, 74

surfaces

adding to gadget file, 229–230

creating "Hello World" app, 4–6

MySpace Open Social, 5

suspension of app, 280

T

tabs, converting to subviews, 245–248

tag templates. ***See*** **inline tag templates, OSML**

tags

CDATA, 225

Data Pipeline, 219–223

OSML, 225–226

TCP (transmission control protocol), three-way handshakes, 119

Template Editor, 83–84

templated apps, 270–271

templates

creating activities using, 80–83

creating with Template Editor, 83–84

defining requestShareApp using, 70–71

inline tag. *See* inline tag templates, OSML

notification built-in, 89

using opensocial.requestShareApp, 85–86

Terms of Use

app suspension from violating, 280

example of app rejection, 270–273

overview of, 266

test pages, REST APIs, 197

testing

managing testers, 279

message bundles, 257

OAuth implementation locally, 166–169

OpenSocial endpoint errors, 300

translations, 259–260

TEXT content type, feed readers, 107–108

TEXT format, partial HTML content, 97

third-party database storage, 64–65

threads, solving readers-writers problem, 51–53

three-way handshakes, 119, 144

Tic-Tac-Toe app. ***See*** **data, getting basic**

time-outs, providing, 300

title, message, 75

TK's apps, 322–324

tools, MySpace development, xxiv–xxvi

TOP_FRIENDS, filtering friend list, 33

TortoiseSVN, 155

transferring data, three-way handshake, 119

translations

creating first message bundle for, 256–257

including and testing, 258–260

internationalization and, 255

translations, *contd.*
message bundle, 257–258
testing, 259–260
transmission control protocol (TCP), three-way handshakes, 119
Try App link, 307–308
TTT namespace
defined, 30
paging, 37
using data, 37–39
TTT.AppDataPlay game object
adding support in game engine, 133–135
adding user feedback, 135–138
clearing game, 144–145
implementing logic flow, 138–144
setting up, 125–133
TTT.Game (game engine), 133–135
TTT.Record.Getter object, 169–172
TTT.Record.Poster object, 169, 172–173
turn-based gaming, 117–133
adding FriendPicker, 119–121
app data game store, 125–133
design overview, 118–119
overview of, 117–118
using FriendPicker, 121–125

U

UI (user interface)
adding FriendPicker to game play, 119–121
building feed reader, 98–101
building using Inspect feature of Firebug, xxv
using FriendPicker, 121–125
UNAUTHORIZED error, cause of, 26
Unicode encoding, and app gadgets, 258
uninstalling apps, 279–280
universal resource identifiers (URIs), REST, 178–179
Updates module, Home page, 68–69
updating
albums and media items in OpenSocial 0.9, 338–339
app data in Open Social 0.8 and 0.9, 341–343
translations, 260
uploading media items, 340–341
URIs (universal resource identifiers), REST, 178–179
URLs
making real MySpace requests, 170–173
warning when placing in attributes, 251
user data
accessing, 11–14
accessing more than just default Profile data, 18–19
accessing Profile information, 15–17
using, 19–24
user interface. *See* UI (user interface)
users
adding feedback to P2P game play, 135–138
data security for, 301
experience/functionality of app, 271–272, 285
implementing logic for selecting opponent, 140
retaining by listening to them, 311
viral spreading and. *See* viral spreading
"user's pick" feed reader, 98
UTF-8 encoding, for app gadgets, 259
utility computing, 298–299

V

validating inputs, 299

variables, setting up AppDataPlay game, 127–128

variables, template

- creating templates with Template Editor, 84
- data types, 80–81
- notifications, 89
- reserved names, 81
- using opensocial.requestShareApp, 86

verify function, OAuth, 157

verify_request function, OAuth, 165–166

vertical scaling, 297–298

videos

- fetching, 42–43
- fetching with get_video function, 193
- fetching with get_videos function, 192–193

View Development Version link, “Hello World” app, 6

Viewer, 236–237

- accessing more than just default Profile data, 18–19
- accessing Profile information, 15–17
- app data store. *See* app data store
- concept of, 9–10
- creating custom tag template definition for, 240–242
- declaring data in gadget XML for, 218–219
- fetching app data for, 342
- fetching friend list for, 30–31
- getting information with os:ViewerRequest tag, 235–236
- os:ViewerRequest tag, 221–222
- permission model for accessing user data, 10–11

viral features and communication, 67–90

- getting app listed on friend updates, 79–88
- sending messages and communications, 74–79
- sending notifications, 88–90
- spreading app to other users, 67–74

viral spreading, 309–316

- of BuddyPoke app, 319–320
- Cubics, 313–314
- Google AdSense, 311–313
- listening to customers, 311
- overview of, 309–311
- Playdom apps using, 325–326
- RockYou! ads, 314–316

Visit Profile link, 7

W

W3C Widgets specification, 260

Watch window, Script tab of Firebug, xxv–xxvi

Web applications, driving performance plateau, 284

Web references. ***See*** **online references**

Web service, 202–203

Westwood, Dave, 318–319

widget formats, commonly known, 260

Widgets specification, W3C, 260

win/loss record

- making MySpace requests, 169–173
- making simple GET request to save, 158–162
- testing OAuth locally, 166–169
- updating with signed POST request, 162–166

Windows Sidebar gadget format, 260

Winloss class, 158–162, 164–166

"Word of the Day", feed reader apps, 110–111

ws.py script, 158, 164–166

X

XHR (XMLHtppRequest) object
- asynchronous JavaScript requests, 15
- external server security constraints for, 91
- makeRequest as wrapper on top of, 94–95
- overview of, 94
- refactoring to build local app data store, 51–52

XML. *See also* gadget XML
- adding feed reader to app, 98
- content type with parsing, 105–107
- handling content for feed reader, 98

XSS (cross-site scripting) attacks, 303

Y

Yahoo! Babelfish, 258

Yue, Dan, 324–326