

The
Pragmatic
Programmers

Facebook Platform Development

with Rails



Michael J Mangino

Edited by Susannah Davidson Pfatzer

The Facets  of Ruby Series

**What people are saying about
*Developing Facebook Platform Applications with Rails.***

Success with Facebook applications is one part idea, one part technical, and one part execution. Mike's book does an amazing job covering all three. He takes you from evaluating your idea and thinking about platform strategy through every step of building the application and then even covers advanced strategies and considerations for scaling. This book will save you a tremendous amount of time learning the platform and building a successful viral application.

► **Keith Schacht**

President, 42 Friends LLC, creators of Growing Gifts

Not only does Mike take the time to explain the technical details required to build a Facebook application, but he also sheds light on important ways to make your app successful, something not often found in a programming book. Mike's book taught me several new tricks that I'm already putting into action to help improve the quality and visibility of my apps.

► **Kyle Slattery**

Lead Social Developer, Viddler

Mike Mangino knows Facebook development. "Sensei Mike's" Karate Poke dojo will teach you what it means to go viral, guiding you through the development of your own Facebook app, and will prepare you for your application's success with discussions about optimization and scaling. He makes Facebook Platform development with Rails as simple as "wax on, wax off."

► **Joseph Annuzzi, Jr.**

CTO, PeerDynamic.com

I had been struggling to integrate Facebook with an existing online game we had built on Rails. Thanks to this book, we were able to create a dedicated Facebook app for GiftTRAP in a couple days. We're really excited to have such a cool viral marketing tool to market our award-winning gift-exchange party game.

► **Nick Kellett**

Inventor of GiftTRAP, gifttrap.com

Developing Facebook Platform Applications with Rails

Michael J. Mangino



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 Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Facebook[®] is a registered trademark of Facebook, Inc. and used by permission. All screen shots of the Facebook Platform and the Facebook web site are copyright Facebook and are used by permission of Facebook. This is not an official guide and was neither created nor endorsed by Facebook.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

<http://www.pragprog.com>

Copyright © 2008 Michael J. Mangino.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-12-3

ISBN-13: 978-1-934356-12-8

Printed on acid-free paper.

P1.0 printing, September 2008

Version: 2009-4-20

Contents

Foreword	10
Acknowledgments	12
Preface	13
Understanding a Successful Facebook Application	14
Developing with Rails	16
About This Book	17
1 Getting Started with the Facebook Platform	20
1.1 Adding the Karate Poke Application	21
1.2 The Parts of a Facebook Application	21
1.3 Getting Inside the App	26
1.4 Setting Up and Running the App	27
1.5 Summary	34
2 Starting Your First Application	35
2.1 Creating a Facebook Rails Application	35
2.2 Sending an Invitation	39
2.3 Giving the Sender Some Feedback	43
2.4 Making Our Invitation Interactive	44
2.5 Updating the Profile	45
2.6 Refactoring to Use Helpers	49
2.7 Summary	51
3 Building the Karate Poke Object Model	52
3.1 Building the User Model	52
3.2 Accessing Facebook from Models	57
3.3 Creating the Move Model	59
3.4 Attack!	60
3.5 Creating the Belt Model	63
3.6 Encouraging Invitations	66
3.7 Getting Data Out of Facebook	67

3.8	Refactoring and Performance	70
3.9	Summary	74
4	Testing Our Facebook Application	75
4.1	Controller Tests	75
4.2	Testing Models	81
4.3	Summary	84
5	Getting Into the Facebook Canvas	85
5.1	Getting Interactive with Forms	85
5.2	Building the Battles Page	91
5.3	Adding Navigation	93
5.4	Hiding Content from Users	97
5.5	Adding Pagination	101
5.6	Adding Some Style	103
5.7	Summary	104
6	Making It More Social	105
6.1	Sending Notifications	105
6.2	Publishing to News Feeds	113
6.3	Comments and Discussion Boards	124
6.4	Spreading by Invitation	128
6.5	Giving the Profile a Makeover	131
6.6	Testing Facebooker Publishers	141
6.7	Summary	142
7	Scripting with FBJS	143
7.1	FBJS Overview	143
7.2	Ajax in FBJS	150
7.3	Summary	157
8	Integrating Your App with Other Websites	158
8.1	Making Content Accessible	158
8.2	Actions That Work Both Ways	160
8.3	Handling Facebook-Specific Data	161
8.4	Sharing Sessions	164
8.5	Accessing Facebook Outside the Canvas	165
8.6	Summary	169

9	Scaling and Performance	170
9.1	Getting Faster with Memcached	170
9.2	Caching Our Views	173
9.3	Caching with refs	177
9.4	API Performance	179
9.5	Summary	185
	Bibliography	186
	Index	187

Foreword

In the early summer of 2007, when Facebook opened its doors to free registrations from the wild and untamed Internet, I remember sarcastically thinking to myself, “Oh, great. Another social network. Just what I need.” I had signed up for a number of social networks in the past and gone through the same routine every time: hundreds of friend requests lead to building out a friend list with which you can accomplish nothing new. I assumed Facebook would be more of the same. But I’m a glutton for punishment, so I signed up anyway.

What I found was remarkably different from the experiences I’d had on other networks. At first, I got friend requests from the usual suspects. We all signed up for the same networks as soon as they were available, and we all looked each other up to connect and try it together. But then, the requests started to ramp up rapidly. People I hadn’t seen or thought about in years started to send me friend requests. Within a couple of days, I had reconnected to people I wouldn’t have thought to even search for. It was exciting and different in that it worked considerably better than previous networks had in the past.

It was then that the Internet collectively realized the existence of a valuable asset that we all started calling the *social graph*. Your social graph is the model and codification of your relationships with other people. These relationships form the basis of the *real* killer app of the Internet. Facebook brought this concept to the forefront by helping users construct a real and interesting social graph more effectively than ever before.

Then it released an API that allowed developers to plug into that powerful social graph management system and create custom applications. So, suddenly we had a massive install base of users, all well connected with their personal circles of friends and colleagues, and we could write applications to operate within this new flourishing ecosystem.

At my workplace, InfoEther, where I am CTO, we'd been focusing on the as-yet-nameless concept of the social graph for a long time. We were (and still are) working on a decentralized social networking platform, and as soon as the Facebook Platform was announced, we knew we had to hook into it. To do that, being Ruby developers, we needed a Ruby Facebook library. Our requirements for this library were that it should

- be written as cleanly and elegantly as possible,
- be pure Ruby with no native extensions,
- be written in idiomatic Ruby style, and
- not depend on any libraries outside the standard Ruby distribution.

Such a library didn't exist, so I wrote Facebooker.

It was shortly after this that I met Mike Mangino at a regional Rails conference in Chicago. He started talking about his own work developing Facebook applications, and I encouraged him to try Facebooker. It was clear from talking to him that, though I had written a Facebook client library from scratch, he had a deeper knowledge and understanding of how Facebook worked than I did. My not-so-secret goal was to coax Mike into codifying his hard-won knowledge in the form of patches and enhancements to Facebooker.

That's exactly what he did. A couple of months later, I received a substantial patch for Facebooker filled with nice enhancements and fixes. I decided to immediately give Mike commit rights to the project, and it wasn't long before he and another open source contributor, Shane Vitarana, effectively took over the maintenance of Facebooker.

When Mike asked me to review this book, I was pleased to have the opportunity but assumed it would be a boring read for me. After all, I had written what is now the de facto standard Facebook library for Ruby and Rails users. Surely, I wouldn't learn anything.

I was very pleasantly surprised (and humbled). Facebook is a powerful and expansive platform, and Mike Mangino is the most expert developer in this platform I know. I learned a lot from reading this book, and I'm happy to be able to add it to my library. If you're doing Facebook development on Rails or otherwise, there is no better resource available.

► **Chad Fowler**

CTO, InfoEther Inc.
Longmont, Colorado

Acknowledgments

This book couldn't have been written without the help of many people. I'd like to thank Roy and Keith, who gave me a reason to start doing Facebook development.

I'd also like to thank my employees. Michael Niessner helped me understand how the Facebook API really works. Jonathan Vaught, Audrey Eschright, and Jeremy Voorhis all helped carry the load while I was spending my days writing. All four of them provided helpful feedback on numerous drafts of this book.

This book builds on top of the excellent Facebooker library from Chad Fowler and Patrick Ewing. They created a truly beautiful and Ruby-like interface to Facebook. Thanks to Shane Vitarana, David Clements, and rest of the Facebooker community for continuing to improve Facebooker.

Thank you to Joseph Annuzzi, Peter Armstrong, Jon Gilbraith, Frederick Ros, Keith Schacht, and Kyle Slattery, who reviewed copies of this book. They provided insightful feedback that smoothed out the rough edges. Special thanks to Charlie O'Keefe for his incredible feedback. Charlie went above and beyond the call of duty, and the end result is much better because of it.

I'd also like to give a very special thank you to Susannah Pfalzer. A first-time author couldn't ask for more in an editor. She was patient and helpful throughout the entire process.

Most important, thank you, Jen, my beautiful wife, for your patience during the many nights filled with writing. I love you and can't put into words how much you mean to me.

Preface

The Facebook Platform offers something for nearly every developer. You might have an idea for the next killer social craze and need a way of getting it in front of a large number of people. Maybe you have an existing application and need a new marketing channel. Even if you just like playing with cutting-edge technology, you'll find the Facebook Platform gives you access to a wealth of information. From detailed information about its users to being easy to develop for the Ruby API, the Facebook Platform has it all.

The Facebook Platform allows you to build applications that take advantage of advanced social features without having to build them yourself. It lets you leverage your users' existing networks while providing you with the tools to achieve rapid growth. Quite simply, the Facebook Platform gets your application in front of more users faster than you can any other way.

It's hard to imagine that this comes from a platform that is barely a year old. Since the platform's release, more than 400,000 people have registered as developers. In that time, more than 24,000 applications have been built, with 140 new applications added every day. Even more incredibly, 95 percent of Facebook's 100 million users have used at least one application built on the Facebook Platform.

So, how do you become one of them? You've taken a great first step. You've decided to build a Facebook Platform application using Rails. Don't be nervous about everything you need to learn. Building a Facebook application is like building any other web application. You build pages in HTML and use JavaScript to make your application more dynamic. You even spend way too long trying to get your CSS to work in Internet Explorer. Once your application is ready to launch, you deploy it to your own servers where it handles HTTP requests like every other Rails application.

Of course, the application you build isn't exactly like any other Rails application. When you build an application for the Facebook Platform, you're not just writing code that uses an API; you're becoming part of Facebook. To your users, your application can look just like any other part of Facebook.

Let's take a look at an example Facebook application and why it is successful.

Understanding a Successful Facebook Application

On any given day, about 500,000 little gifts are sent by the users of Growing Gifts, an application created by Keith Schacht. With Growing Gifts, shown in Figure 1, on the following page, you can send a flower to any of your Facebook friends. The flower you send starts as just a sprout and grows over four days in the recipient's profile.

This seems like a simple little application, but it demonstrates a serious point about Facebook. Because it is so simple to install and use applications, your application doesn't have to provide nearly as much value as it would outside Facebook. Facebook has reduced the cost of using an application.

Let's look at how Growing Gifts would work outside Facebook. If you saw a gift on your friend's website, you could click it and be taken to the Growing Gift site. Once there, you would need to create an account and verify your email address. Next, you would need to find and enter the email addresses of your friends. Your friends would then receive an email and go through the same process of signing up for an account. Finally, they get to see their flower. They would need to return to the Growing Gift website every day to watch their gift grow.

Now let's look at the typical case for a new user sending a gift via Facebook. If you see a gift you like in your friend's profile, you can click the gift. You are then prompted to authorize the Growing Gifts application. You click Allow and are taken to a screen where you can send other gifts to your friends. All you need to do is select a gift and then start typing your friends' names. Facebook even uses autocomplete to make friend selection easy. Once you've selected a few friends, you can just hit the Send button. Every day, when they return to Facebook, your friends can see how your gifts have grown. That's all it takes to give your friends a little joy. Can you see why growing gifts is such a hit on Facebook but would never work as a stand-alone site?

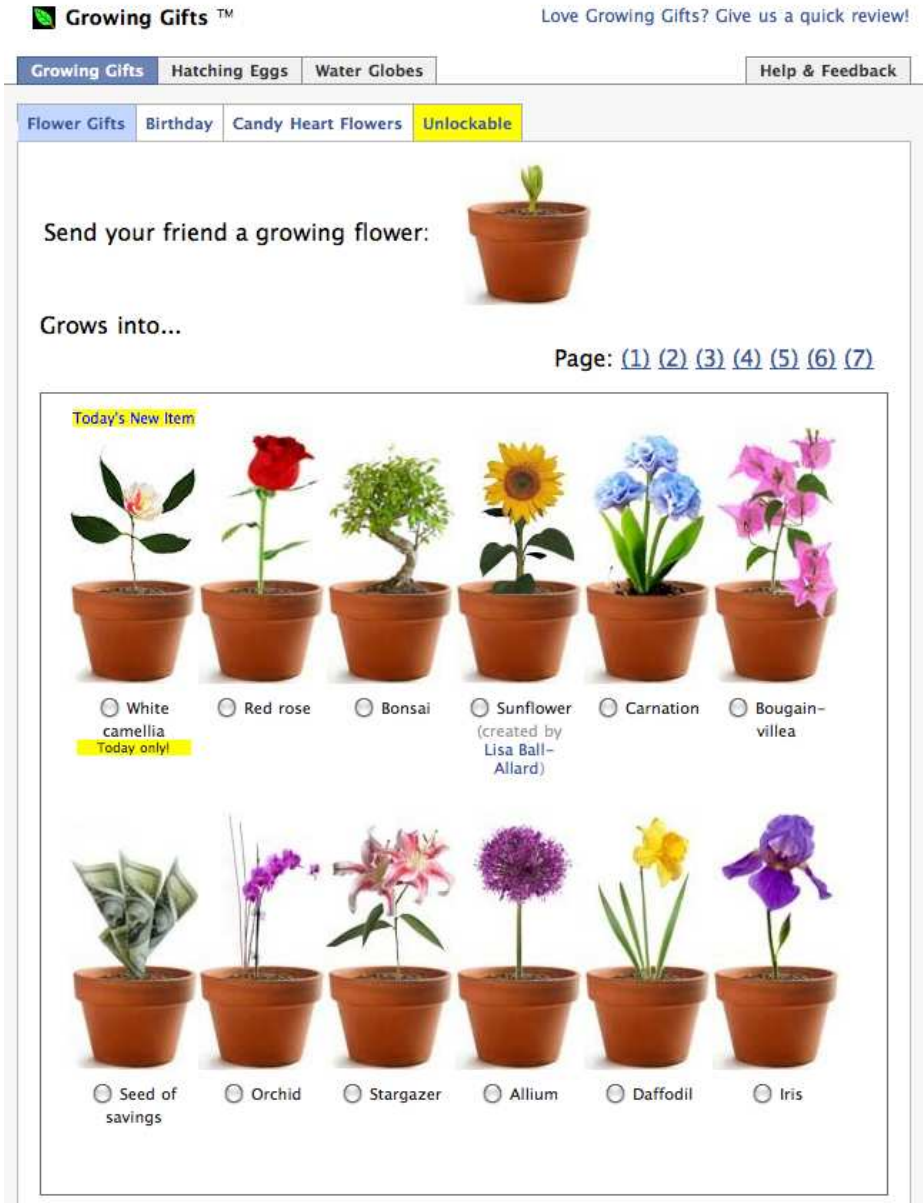


Figure 1: Growing Gifts shows the power of a simple social application.

By making applications easier to find and use, Facebook has opened the door to a whole new class of application. Things that were too costly before are suddenly successful. By cost, I don't mean just the price of the application; I mean the cost of your time and effort. In economics, this is called the *opportunity cost* of your time. After all, I'm sure you have something better to do than to create yet another account for a web application.

The tips we'll learn in this book were used to build Growing Gifts, which is built using Rails. In fact, many of the features in the Ruby library for Facebook, as well as the advanced Facebook techniques discussed in Chapter 9, *Scaling and Performance*, on page 170, come from Growing Gifts because Growing Gifts is a hugely successful application. With more than 5 million users, Growing Gifts can teach us a lot about how to build a compelling Facebook application.

Developing with Rails

We're going to do more than just learn how to build a Facebook application. Specifically, we're going to learn how to build a Facebook application with Ruby on Rails version 2.0. All the examples should work on Rails 2.1 as well. There are some very compelling reasons to develop your Facebook Platform application with Ruby on Rails. It all starts with *Facebooker*.

Facebooker, a project originally started by Chad Fowler, has grown to be an almost complete Ruby implementation of the Facebook API. Facebooker is not the first Ruby interface to Facebook—that honor belongs to *RFacebook*. RFacebook did not include much Rails integration and is no longer supported by its author. Facebooker is, however, the first library with deep Rails support. Facebooker was designed to make developing a Facebook application feel like building any other Rails application.

With Facebooker, you get access to all the great Rails features that you're used to using. You can store information in a user's session and have it persisted between requests. You can easily use Ajax to make your application more dynamic. You can define your database schema using migrations.

On top of all this, you'll also get some Facebook-specific features. You'll see a suite of view helpers to make integrating with Facebook a snap in Chapter 5, *Getting Into the Facebook Canvas*, on page 85. Facebooker

will provide you with an ActionMailer-like way of sending messages through Facebook, which is covered in Chapter 6, *Making It More Social*, on page 105. Finally, Facebooker makes it painless to turn your existing application into a Facebook application, as we'll see in Chapter 8, *Integrating Your App with Other Websites*, on page 158.

By developing your Facebook application with Ruby on Rails, you're getting the best of both worlds. You're using a powerful and productive framework to build an application for an incredible platform.

About This Book

In this book, we'll do more than just talk about the Facebook Platform. We'll build a real application called Karate Poke, which is a game that allows you to attack your friends. We'll go on a tour of the API and build features that use the majority of the platform. Once we have a working application, we'll look at the tips and tricks you can use to help your application handle large amounts of traffic.

You'll get more than just an introduction to Facebook development. You'll also get the complete code to the Karate Poke¹ application. The code for each chapter is available from the book's website.² Accessing the code is even easier if you have the PDF. Just click the gray box before each code snippet to view the file in your browser.

What You Should Know

We'll build this application from the ground up, so you don't need any previous Facebook development experience. You will need to have a Facebook account, and it would be helpful to have used Facebook briefly. As we build our application, we'll look at the various parts of Facebook both as a developer and as a user.

Although you don't need Facebook development experience, you will need some Rails development experience. You don't need to be a Rails expert, but it would be beneficial to have written at least one application. If you are new to Rails, I recommend purchasing *Agile Web Development with Rails* [TH05] and working through it as you go.

1. <http://apps.facebook.com/karatepoke>

2. <http://www.pragprog.com/titles/mmfacer>

How the Book Is Structured

As I mentioned earlier, our goal in this book is to build a Facebook application. Along the way, we will take a tour of the Facebook API. We'll start simple and become more advanced as we go.

We'll start our tour by looking at the different parts of a Facebook application. Next, we'll get our development environment up and running. From there, we'll look at some basic Facebook functionality before building the back end of our application. Next, we'll learn some Facebook-specific tricks to make testing easier. Once we have our basic model in place, we'll turn our attention to some very Facebook-specific concepts, starting with the Facebook Markup Language (FBML). Along the way, we'll look at how we can make our application more social before finishing with a look at performance.

This book is not meant to be an exhaustive reference. Instead, it is meant to be a tutorial that gets you up to speed quickly on the Facebook Platform. If you are looking for detailed API information, you can find it on the developer website.³ Throughout the book, I will link to the developer documentation for features that are lightly covered.

Even though this isn't an exhaustive reference, we will look at most of the Facebook API while building our application. We will cover all the most important features of Facebook Platform development. As we go, we'll focus on why we build our application this way and how all the pieces fit together. We won't touch on any of the business issues involved, such as how to make money from our application. For the most part, those issues are the same as any other website.

Aiming for a Moving Target

I would love for this book to be completely accurate forever, but unfortunately, we're tracking a moving target. Facebook released sweeping changes to the way the platform works in the summer of 2008. This book has been completely updated to include up-to-date descriptions of the new API elements and new integration strategies. I'm sure next year will be much the same. As such, occasional inaccuracies will creep in. To stay up-to-date, check out the forums for this book as well as its Facebook page.⁴

3. <http://developer.facebook.com/documentation.php>

4. <http://forums.pragprog.com/forums/59> and <http://www.facebook.com/pages/fpdwr/12146405638>, respectively

Part of being a Facebook developer is embracing change. Facebook maintains a news site for developers that contains advance notice of changes.⁵ Facebook also provides a feed of information about changes made by each weekly release.⁶ As a developer, you should subscribe to both feeds. Although most changes give at least a week or two of advance notice, Facebook does make the occasional change with no warning.

Changes will not be limited to just Facebook. The Facebooker library used throughout this book is relatively new and will continue to evolve over time. For the most up-to-date documentation, see the Facebooker website.⁷

Thanks for joining me on this tour. If you have questions, you can ask them on this book's forum.⁸ If you build a really cool application that you want to share with the world, post that there too. We have a lot to do together, so let's get started.

5. <http://developer.facebook.com/news.php>

6. http://www.facebook.com/feeds/api_messages.php

7. <http://facebooker.rubyforge.org>

8. <http://forums.pragprog.com/forums/59>

Chapter 1

Getting Started with the Facebook Platform

Congratulations on deciding to build a Facebook application using Rails. You're about to embark on an exciting journey. It'll be a fun journey with some interesting stops along the way, because you will be in the driver's seat. You'll learn the basics of Facebook application development by building a complete Facebook application throughout this book.

The application we'll build is called Karate Poke. It's a simple game where you battle your friends and other users. As you progress through the game, you'll earn karate belts and learn new moves. Although Karate Poke may not be identical to the Facebook applications you'll want to build, it serves as a good introduction. It's a small application, so we'll be able to build the whole thing as we go. It also uses almost the entire Facebook API. We'll see how to build invitations and notifications. We'll use most of the UI elements that Facebook provides. We'll even be able to use some advanced performance-tuning techniques to make it handle the demands of millions of users.

In this chapter, we'll start out by taking a tour of Karate Poke. After we've seen the basics of Karate Poke, we'll create an application using the Facebook Developer tool. Next, you'll set up your computer to run a prebuilt test application. We'll finish up by creating a few test users. By the time we've finished this chapter, you'll have gotten your feet wet with Facebook development and will be ready to start coding. We have a lot to do, so let's get started.

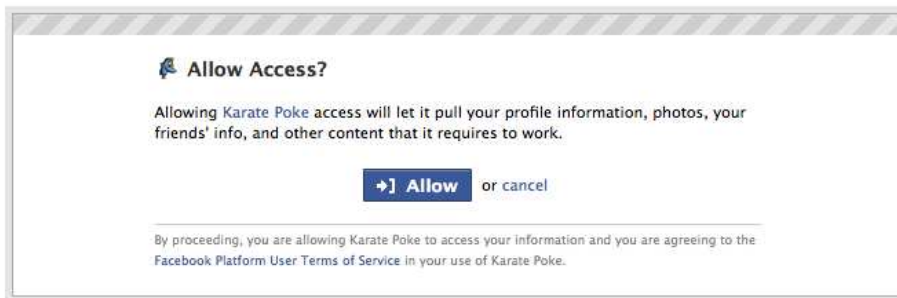


Figure 1.1: Facebook asks you to authorize an application before you use it.

1.1 Adding the Karate Poke Application

Open your browser, and go to the URL¹ for Karate Poke. You're seeing the application authorization page shown in Figure 1.1. You must authorize an application before it can get access to any of your information. This is the first of several privacy features that are part of the Facebook Platform.

When you authorize an application, you are giving it access to some of your personal data. You allow that application to access your profile information and your list of friends and even to send updates about the activities you've performed. Along with these permissions, you can elect to give an application the ability to do more on your behalf such as add a box to your profile or send you email.

These permissions aren't set in stone. You can change them and even deauthorize an application at any time. As an application developer, you'll need to be aware of this. It's important to make sure your application degrades nicely when a user limits its capabilities.

1.2 The Parts of a Facebook Application

Now click Allow to authorize the application. Once you've done that, you will be taken to the *canvas page* of Karate Poke. That seems like a good place to start our tour of the three main parts of a Facebook application: the canvas, the profile area, and messages.

1. http://apps.facebook.com/karate_poke

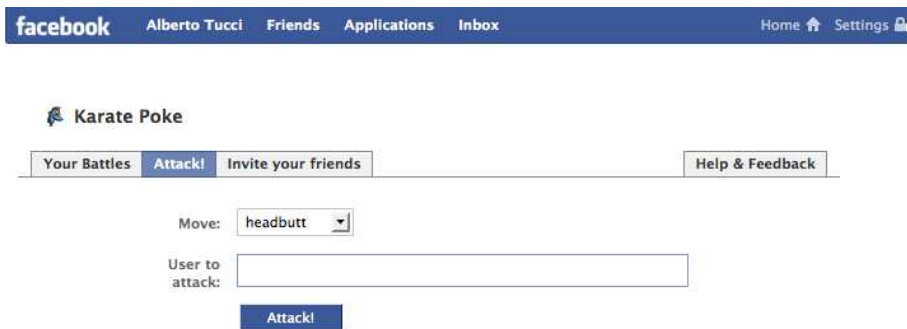


Figure 1.2: The canvas is where users interact with your application.

The Facebook Canvas

The canvas page is the main place where your users interact with your application. You'll notice that it looks like any other Facebook page, including the navigation area along the top. Facebook actually inserts your application right into the middle of the page, just like a Rails layout. You can see what this looks like in Figure 1.2.

We'll cover the Facebook canvas in detail in Chapter 5, *Getting Into the Facebook Canvas*, on page 85. You don't need to know much about the canvas yet, though. For the purpose of this demo, just go ahead and try attacking one of your friends. You can select any move from the pull-down menu. Once you've picked a move, start typing a friend's name into the text field. Isn't that typeahead cool? That's just one line of code!

Select any of your friends, and then click the Attack! button. Karate Poke will let them know you've attacked them. You should be taken to your battle history page where you'll see the result of your attack. On this page, you should now see an Add to Profile button. Click that, and your battle record will be added to your profile.

The Profile Page

Click your name in the top navigation. Once your profile loads, click the Boxes navigation link. At the bottom of the page you should see a box from Karate Poke. Every Facebook application you install gets access to a little box in your profile.



Figure 1.3: Applications can add data to a user's profile.

You can see an example of this in Figure 1.3. The profile area is a place to show off information about its owner. For instance, my Karate Poke profile box shows my battle history. My Growing Gift profile box shows gifts that my friends have sent me. If a user wants, they can feature your application more prominently on the front page of their profile. They can even give your application its own tab. We'll see how this works in Chapter 6, *Making It More Social*, on page 105.

Facebook doesn't limit the amount of information an application can write to a user's profile, but it does limit the content. Facebook allows advertising in the canvas but not in the profile.

Now that we've seen the canvas and profile, let's move on to the last major part of a Facebook application.

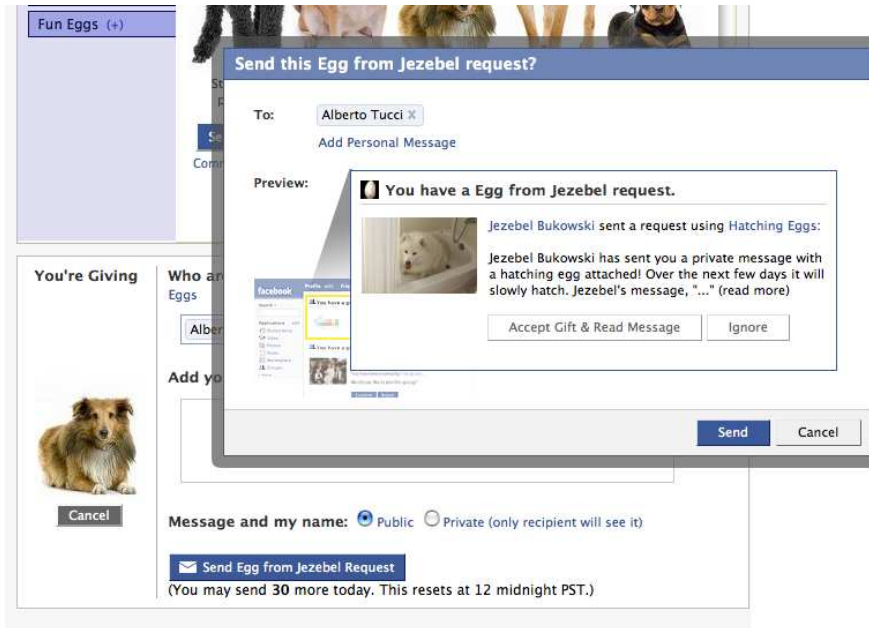


Figure 1.4: Facebook makes you approve a request before it is sent.

Messages

Along with giving you access to the canvas and profile, the Facebook Platform also gives you several different types of messages. Your application can send *requests*, *notifications*, and *news feed* items. This may seem like a lot of different message types, but they all fill very different needs. If you're a Facebook veteran, you've probably used all these message types without even noticing.

A request is exactly what its name implies—a request to perform an action. When somebody asks to be your friend, that's a request. A request is just a message with buttons for performing actions. Your application can send requests on behalf of a user but only to their friends. Users must approve requests before they are sent. You can see the approval process in Figure 1.4.

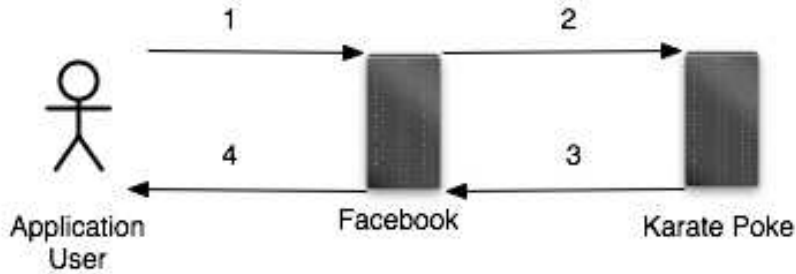
Notifications are similar to requests but lack the call to action. Your application can send notifications on behalf of the user without requiring their interaction. Unlike a request that can be sent only to a user's friends, notifications can be sent to any user of your application. Face-



Figure 1.5: The Facebook feed shows relevant information from your friends.

book limits the number of notifications that an application can send for a user in a given day. Users can choose to block notifications from certain applications and can even mark a notification as spam. Be careful sending notifications. If enough users mark your notifications as spam, you'll lose the ability to send notifications for thirty days.

News feed items are the last form of communication available to the application developer. News feed items are what populate your wall page. They tell your friends a little bit about what you've been doing. Your friends' news feed items are shown on your home page. Your application can send quite a large number of news feed items each day, but they aren't guaranteed to be visible. Facebook tries to decide what is interesting for each user and shows only a subset of the available items, as you can see in Figure 1.5.



1) A user requests a page from Facebook. 2) Facebook sends the request to your server. 3) You send a response to Facebook. 4) Facebook processes your response and puts your page into its layout and sends it back to the user.

Figure 1.6: The path of a Facebook HTTP request

1.3 Getting Inside the App

Let's dig in a little deeper to understand how your application actually becomes part of Facebook. I'm going to start with the canvas since it will probably be the largest portion of your application. Building the canvas portion of your Facebook application is similar to any other website. Your web server will receive requests and send back pages. Of course, it's a little different since your page ends up inside a Facebook page.

When a user requests a page from Facebook, Facebook passes that request on to your application. Your application responds with your content. Facebook then puts your page into the middle of theirs and sends it back to the browser. You can see a picture of this in Figure 1.6.

This all happens seamlessly from your users' perspective. In fact, it's pretty seamless from your standpoint as well. For the most part, you can forget about Facebook standing between you and the user. Sure, it's not exactly like regular development, but it's pretty close.

This may seem like a complex architecture, but Facebook has a good reason for the complexity. By acting as a middleman, Facebook can process the page you send back to the user. This allows Facebook to insert your page into its layout and also to provide you with some really

powerful tools. We'll look at this in detail in Chapter 5, *Getting Into the Facebook Canvas*, on page 85.

Things are different for the profile. I have ten applications installed on my profile page. If each application took even a second to respond, it would take ten seconds to load my profile! To prevent long profile load times, Facebook caches profile information. You can update it at any time by just sending Facebook the new content to display. This can feel a little strange at first, but you'll be happy to avoid the millions of page views you'll get if your application develops a large following.

1.4 Setting Up and Running the App

Now that we've looked at the parts of a Facebook application, let's actually create one. Before we write any code, however, we're going to need to do some configuration. We'll start by using the Facebook Developer tool to set up our new application. Once we've done that, we will run a test application to make sure everything is in working order.

Creating a Facebook Application

First, you need to install the Facebook Developer application. If you don't already have a Facebook account, create one now. Next, go to the main Developer application URL,² and install the Developer application. The Developer application lets you configure and manage your Facebook Platform applications. Once you have it installed, its icon should appear in the Applications menu in the top navigation bar of every Facebook page.

Let's take a brief look at the Developer application. To open it, click Developer in the application list. You will be taken to its canvas page where you should see something that looks like Figure 1.7, on the following page. From here, you can create new applications, edit existing applications, and view basic statistics. The lower section of the page includes a summary of recent developer forum posts and a news feed from Facebook.

To create an application, click the Set Up New Application button in the upper right. Facebook will send you to a screen where you can enter the application name. The application name you choose here is displayed in the application directory and is used in messages sent by

2. <http://www.facebook.com/developers>

Figure 1.7: We'll use the Developer application to create and configure Facebook applications.

Figure 1.8: You've now created your first Facebook application.

your application. You can change it at any time, so just create a fun name. For applications I'm working on, I typically use my name as the application name.

Once you've agreed to the terms of service, click the Submit button to create your application.

Configuring Our Network

You've just created your first Facebook application! You should now see an entry for your application like the one in Figure 1.8, on the previous page. This summary page lists basic information about each application you've created. You can return to this page at any time by clicking the See My Apps link in the upper right of the main Developer page.

Before we can configure our application, we'll need to configure our network. During normal Rails development, you run `script/server` on your machine. Your browser requests pages locally without your requests ever leaving your computer. You can do all your development without a network connection. In Facebook development, you request pages from Facebook, which then sends requests to your computer. This means that while you are developing your application, Facebook will need to talk to your computer. If you develop on a laptop like I do, this means you need to make your local web server available to the world at large.

If you're working where your computer is directly connected to the Internet, such as with a DSL or cable modem connection, you can just open your firewall to allow connections on port 3000. (You are using firewall software, aren't you?) If you are behind a router, you will need to configure port forwarding to send requests to your machine. Check your router manual to find out how to do this. If you control your own network, set that up now, and skip ahead to Section 1.4, *Configuring Our New Facebook Application*, on the following page. If you don't control your own network, things get a little trickier.

If you have access to a machine directly connected to the Internet, you can use `ssh` to send requests to your local machine.³ For example, I have a machine named `mingus.example.com` that is on the Internet. I can do Facebook development from a coffee shop by running this:

```
$ ssh mingus.example.com -R :3001:127.0.0.1:3000 sleep 99999
```

This tells `ssh` to take port 3001 on `mingus.example.com` and forward it to port 3000 on my local machine. I set my callback URL to be `http://mingus.example.com:3001`. It's important to include the first colon in the `-R` argument. That tells `ssh` to listen on all interfaces. If you leave that off, Facebook will not be able to connect to `mingus`. You also need Gate-

3. If you don't have access to a properly configured server, you can use `Tunnlr` (<http://tunnlr.com>) instead.

wayPorts set to clientspecified in the sshd_config file on the server to which you are connecting.

That’s a lot to remember. Instead of typing that cryptic command every time you want to start a tunnel, you can use the rake tasks built into Facebooker, the Ruby Facebook Platform development library. You’ll need to set the parameters in the tunnel section of your facebook.yml file. For the previous example, I would use this:

```
development:
  ...tunnel:
    public_host_username: mmangino
    public_host: mingus.example.com
    public_port: 3001
    local_port: 3000
```

With that in place, you can simply run rake facebooker:tunnel:start to set up your tunnel.

If you work for a company that blocks ssh tunneling, you may be able to use a DMZ proxy setup. To do this, you run a proxy server, such as Squid or Apache, on a machine that is directly connected to the Internet. You can have that machine send requests to your computer. You may need to talk to your IT department to make this work.

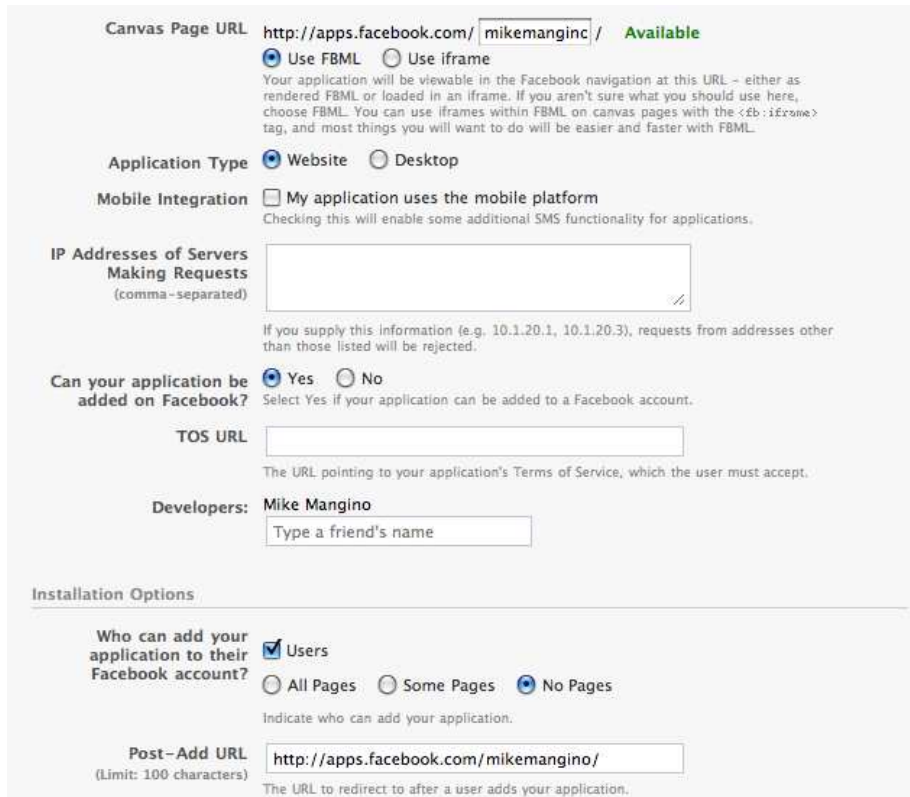
If these options don’t work at your location, you could always try going to a local coffee shop and developing from there!

Configuring Our New Facebook Application

We’re going to start by configuring Facebook to talk to a prebuilt application. Click the edit settings page of your newly created application. The edit settings page contains a huge number of options. Luckily, we need to configure only four options. You’ve already set your application name, so we can ignore that. You’ll also need to set a “callback URL.” This is the URL Facebook uses to request pages from your application.

Now that we have our network configured, we can set up our callback URL. You’ll need to know your IP address to tell Facebook how to talk to you. There are many services⁴ that will provide this information. From your application settings page, enter `http://«your_ip»:3000/` into the Callback URL field where «your_ip» is your IP address. It is critically important that you include the trailing slash in your callback URL.

4. One example is <http://whatsmyip.org>.



The screenshot shows the Facebook application configuration interface. The 'Canvas Page URL' is set to 'http://apps.facebook.com/mikemanginc/' and is marked as 'Available'. The 'Application Type' is set to 'Website'. Under 'Mobile Integration', the checkbox 'My application uses the mobile platform' is unchecked. The 'IP Addresses of Servers Making Requests' field is empty. The question 'Can your application be added on Facebook?' has 'Yes' selected. The 'TOS URL' field is empty. The 'Developers' section shows 'Mike Mangino' as the developer, with a text input field for adding more developers. Under 'Installation Options', 'Who can add your application to their Facebook account?' has 'Users' selected. The 'Post-Add URL' is set to 'http://apps.facebook.com/mikemangino/'.

Figure 1.9: We need to configure our application to allow installation.

If you do not, Facebook will have problems sending requests to your application.

Once that is done, you need to pick a path for the canvas page URL. All applications on Facebook are available at `http://apps.facebook.com/«canvas_path»`. The «*canvas_path*» that you pick must be unique. What you choose will be visible to all your users but doesn't have any importance beyond aesthetic value.

We'll need to allow our application to be added by Facebook users. To do that, click Optional Fields to show some additional options. Click Yes next to "Can your application be added on Facebook?" and select Users next to "Who can add your application to their Facebook account?" Finally, enter your Facebook canvas page URL into the Post-Add URL

field. You can see what this looks like in Figure 1.9, on the previous page. Save, and you will be returned to your application listing.

Configuring the Rails Application

Now you just need a Rails application! Download the code for this book,⁵ and then go to the `network_test` directory. You need to edit `config/facebooker.yml` to tell it about your Facebook application. The API key and secret key act as a username and password for your application. We'll look at them in more detail in Section 2.1, *The Details of Facebook Signatures*, on page 37. Using your IP address and the canvas path you set previously along with the API key and secret key, make the development section of your `facebooker.yml` file look like this:

```
development:
  api_key: «api_key»
  secret_key: «secret_key»
  canvas_page_name: «canvas_path»
  callback_url: http://«your_ip»:3000
```

As an example, here is my `facebooker.yml` file:

```
development:
  api_key: e34ec276c93b8b443fd15691c57908c5
  secret_key: secret
  canvas_page_name: mangino
  callback_url: http://web1.tunnlr.com:10103
```

Once that is done, you can run `script/server` to start your web server. With your web server running, let's fire up a web browser and point it to your server.⁶ You should see a "Hello from Rails" page. If you don't, you should double-check your network setup. If it works, try going to your application on Facebook.⁷ If you see a "Hello from Facebook" message, you're all set!

Dealing with Errors

Of course, things don't always work the first time. You'll see a couple of error messages a lot when working with Facebook. If you see "The URL `http://«ip»/«canvas_path»` did not respond," that means Facebook couldn't talk to your web server. Usually, this happens because you

5. The code is available at http://www.pragprog.com/titles/mmfacer/source_code.

6. `http://«your_ip»:3000/network_test`

7. `http://apps.facebook.com/«canvas_path»`

forgot to run `script/server` or because your network isn't properly configured. If you did start the server, make sure you can reach your web server from outside your network.

If your application raises an exception, the error message will be shown inside the Facebook canvas in development mode. If you see something like `Facebooker::Session::IncorrectSignature`, it means you configured your `facebooker.yml` file incorrectly. The API key and secret key are like a username and password that you use to authenticate yourself to Facebook. They not only allow Facebook to know that it is talking to you, but they also allow you to verify that the request you are processing came from Facebook.

This is tricky stuff, so don't feel bad if it takes you a little while to get everything working. I've set up a Facebook page⁸ to help you get up and running. It has an up-to-date list of errata as well as discussion forums. If you get stuck, feel free to ask for help. That's why I created the page.

Creating Test Accounts

Congratulations! You're now running a Facebook application. In the next chapter, you'll start building your own application. Before you do that, let's take care of a couple of housekeeping issues. Since Facebook applications are meant to be social, you will need more than one user account to test some features. You can create several test accounts to use before your applications are ready to be released.

Test accounts are very similar to regular accounts but come with a few benefits. First, a Facebook test account can't be friends with a regular user account. That means there is no danger you will accidentally send a message to one of your friends while testing your application. Additionally, Facebook won't remove your test account for being a fake account like it will for regular user accounts. Finally, using a test account can help keep your test application from leaking to the general public. Before Facebook created test accounts, one of my employees accidentally had his test application installed by more than 1,000 users. It was really starting to slow down his laptop!

Let's create a test user. Test accounts start as regular users, so you'll need another Facebook account. To create one, you'll need another

8. <http://www.facebook.com/pages/fpdwr/12146405638>

email address. If you don't have an unlimited supply of email addresses, you can sign up with a free provider like Yahoo or Gmail. Once you have an email address, create a new Facebook user. Log in to your new Facebook account, and go to the test account creation URL.⁹ Your account will now be a test account. You will probably want to create at least three test accounts so that you can test actions between friends and actions between nonfriends.

You can create boring names for your test users, like Joe Smith I and Joe Smith II, or you can have a little more fun and create some alter egos. I like to come up with creative names and profile pictures for my test users.

Along with using test accounts, you can also restrict your application to be visible only to developers. If you check the Developer Mode checkbox, your application will be installable only by developers. This effectively limits the spread of your application to just a small number of people. Unfortunately, if you select this option, your test accounts will not be able to install your app.

1.5 Summary

Look at all you accomplished in this chapter! You installed the Facebook Developer application (and maybe created a Facebook account in the process). You configured your network and actually ran an application locally. You created some test user accounts so that you don't accidentally send strange Facebook messages to your boss. That's all well and good, but let's get on with the coding!

9. http://www.facebook.com/developers/become_test_account.php

Chapter 2

Starting Your First Application

So far, we've used Facebook's Developer tool to create keys for an application. We've also configured Facebook to talk to our computer and to run a prebuilt application. After all that setup, let's get started writing Karate Poke.

In this chapter, we're going to add two features found in many social networking applications. We'll start by building an invitation system to allow our users to tell their friends about Karate Poke. From there, we'll add some content to our users' profiles.

Before we do that, though, let me explain what the "Poke" in Karate Poke means. Poke is a very simple Facebook application that allows you to let another user know you're thinking about them. When you poke a user, they just receive a message telling them that they were poked by you. Because of the simplicity of the application, developers quickly created a number of variations on the poke concept. Poke applications like Super Poke and X Me are some of the most popular applications on Facebook.

2.1 Creating a Facebook Rails Application

In the previous chapter, we configured an application with the Developer tool. Now we're going to write some code for Karate Poke.

Configuring Rails

First, we're going to need a Rails application. Let's create an application called `karate_poke`:

```
$ rails karate_poke  
...
```

Now that we have a Rails application, let's turn it into a Facebook application. We'll do that by installing the Facebooker plug-in. Facebooker is the Ruby library that knows how to talk to Facebook; we discussed it in the preface. It provides access to the Facebook API, some view helpers, and the glue that makes our application part of Facebook. The development of Facebooker is hosted by GitHub.¹ Git is a distributed version control system that was designed to allow a large number of users to contribute to a project. Starting with Rails 2.1, plug-ins can be directly installed from Git repositories as long as you have Git installed.²

Run `script/plugin install git://github.com/mmangino/facebooker.git` to install Facebooker. If you don't have Git installed, you can download a compressed archive of the plug-in and unpack it into your `vendor/plugins` directory.³

If you are using a pre-2.1 version of Rails, you will also need to install the `json` gem by running `gem install json`. If you are following along on Windows, you should use the `json-pure` gem instead.

Next, we'll need to do a little configuration to our Facebook application. This next step should look familiar to you. Installing Facebooker will create `config/facebooker.yml`. Open this file, and fill in the development section. You can use the API key and secret key that came from the application you created in Section 1.4, *Configuring the Rails Application*, on page 32, or you can follow the same steps and create a new application. I'm lazy, so I'm going to use my existing Facebook application.

Since you've had to do this step twice already, you've probably guessed that these lines of code are important to a Facebook application. Let's look at what they actually do.

The first two lines are our application's username and password. They authenticate our application to Facebook. Just as importantly, they let our application verify that requests are coming from Facebook. It sounds odd that we need to verify that requests come from Facebook. I'll explain why in Section 2.1, *The Details of Facebook Signatures*, on the following page.

1. The main repository page is available at <http://github.com/mmangino/facebooker/tree/master>. You can find the API documentation at <http://facebooker.rubyforge.org>.

2. Git is available for all major platforms at <http://git.or.cz/#download>.

3. The plug-in can be downloaded from <http://github.com/mmangino/facebooker/tarball/master> in `.tar.gz` format or from <http://github.com/mmangino/facebooker/zipball/master> in `.zip` format.

The next line tells Facebooker what to use for our application's canvas path. We talked about the canvas path in Section 1.4, *Configuring Our New Facebook Application*, on page 30. Facebooker will automatically include our canvas path in all our application's links.

The last line is used to tell Facebooker where to find our server. Even though all requests for our canvas pages go through Facebook, our images will be requested directly from our server. By setting the `callback_url` parameter, Rails knows to use the hostname of our server instead of `apps.facebook.com`.

The Details of Facebook Signatures

One thing we won't do in this book is build a login controller. We can depend on Facebook to handle authentication for us. In fact, Facebook sends us the ID of the current user and their whole list of friends on every request. That makes our life quite a bit easier. It also can cause some security problems.

In typical web development, your application would never include a logged-in user's ID as part of the URL. After all, a malicious user could change the user ID in the URL to access your application as a different user. Facebook development is a little different.

In Facebook development, we never talk to our users directly. All requests come from Facebook. To make sure this is the case, we can verify the signature that is sent by Facebook on every request. A digital signature is a way to use cryptography to verify that something actually came from the person who it appears to be from.⁴ Facebook sends a number of parameters that start with `fb_sig`. All these parameters are used in the signature validation.

When Facebook sends our applications a request, it builds a string that includes all the `fb_sig` parameters in alphabetical order. It then adds our secret key to the end of that string and calculates the MD5 sum.⁵ Facebook then adds this signature to the request in the `fb_sig` parameter. When Facebooker receives a request, it goes through the same steps to recalculate the signature. If the value that Facebooker calculates matches the one in our request, it proves that the request came from somebody who knows our secret key. We also know that the

4. For more information, see http://en.wikipedia.org/wiki/Digital_signature.

5. MD5 is a cryptographic one-way hash function. You can learn more at <http://en.wikipedia.org/wiki/MD5>.

request wasn't changed after it was sent; if the request was changed, the signatures wouldn't match. If they don't match, then we know that the key the sender used to calculate the signature doesn't match our key. When that happens, Facebooker will raise an exception. You will probably see this exception from time to time. It doesn't normally come from a forged request. It normally shows up when you have the wrong key in your facebooker.yml file.

Setting Up the Controllers

Now that we have Rails configured, we need to do a little setup in the application controller. Facebook limits the information we can learn about users who haven't authorized our application. To simplify development, let's start out by requiring that all of our users give us access to their information. We'll see (in Section 6.2, *Making Our Application More Visible*, on page 117) how we can make our application visible without requiring a user to allow access.

To require a user to allow access, we will need to call a method provided by Facebooker to our application controller:

```
Download chapter2/karate_poke/app/controllers/application.rb
```

```
class ApplicationController < ActionController::Base
  helper :all # include all helpers, all the time

  # See ActionController::RequestForgeryProtection for details
  # Uncomment the :secret if you're not using the cookie session store
  protect_from_forgery :secret => 'a7cabcdf1499df9ded55d8a3797d9387'
  ensure_authenticated_to_facebook
end
```

By default, Rails uses browser cookies to store session information. This used to cause problems because early versions of the Facebook Platform didn't support cookies. Facebook recently fixed this issue, and now Rails sessions work out of the box for users who have allowed access. Facebook doesn't provide cookie support or any other kind of tracking for users who haven't allowed access to our application.

We have just one more change to make before we're ready to go. Starting with version 2.0, Rails included a feature to stop cross-site forgery attacks.⁶ Unfortunately, this feature doesn't play well with Facebook. We'll need to disable it by editing `environment.rb` and setting `allow_forgery_protection` to `false`, as shown here:

6. You can read about this at <http://ryandaigle.com/articles/2007/9/24/what-s-new-in-edge-rails-better-cross-site-request-forging-pr>

```
Download chapter2/karate_poke/config/environment.rb
```

```
Rails::Initializer.run do |config|  
  config.action_controller.allow_forgery_protection = false
```

Our setup is now done, and we're ready to start coding.

2.2 Sending an Invitation

Now that we've configured our application, it's time to actually write code. We're going to start by building an invitation system. Invitations are one of the top tools our application can use to grow virally, discussed in the sidebar on page 46. Let's start our implementation by creating the Invitations controller.

Creating the Invitations Controller

We can create our controller using the Rails generator:

```
$ script/generate controller invitations
```

We're going to use the new RESTful⁷ features of Rails to build Karate Poke. REST stands for Representational State Transfer. Starting with version 1.2, Rails provides support for creating applications using REST principles. These conventions tell us that we should use the new action to show a form. Before we do that, let's set up our routes. To do that, we'll edit the default config/routes.rb file to make it look like this:

```
ActionController::Routing::Routes.draw do |map|  
  
  # Tell Rails to make invitations into a resource  
  map.resources :invitations  
  # Use our new method as the default page  
  map.root :controller=>"invitations", :action=>"new"  
  
  # Install the default route as the lowest priority.  
  map.connect ':controller/:action/:id.:format'  
  map.connect ':controller/:action/:id'  
end
```

With that little bit of setup out of the way, we can move on to the user interface.

7. If you haven't used the new RESTful Rails features, Geoffrey Grossenbach provides a great overview at <http://peepcode.com/products/restful-rails>.

Creating the Invitation Form Using FBML

We'll start building our user interface by creating a simple view:

```
Download chapter2/karate_poke/app/views/invitations/new.erb
```

```
<fb:fbml>
  <fb:request-form
    action="<%=new_invitation_path%>"
    method="POST"
    invite="true"
    type="Karate Poke"
    content="I added a cool application." >
    <fb:multi-friend-selector
      showborder="false"
      actiontext="Invite your friends to use Karate Poke." />
  </fb:request-form>
</fb:fbml>
```

That looks a little like HTML, but I bet you've never seen those tags before. What you're seeing is the Facebook Markup Language (FBML). FBML is one of the most powerful features of the Facebook platform. It acts as an extension to HTML that gives you some prebuilt user interface components. FBML is translated into normal HTML when Facebook processes your page. We'll see a little bit of FBML here and will cover it in great detail in Chapter 5, *Getting Into the Facebook Canvas*, on page 85.

Before we get into the details of how the view works, let's see what it does. Start up `script/server`, and then open a browser to your invitation page.⁸ Make sure you're logged in as one of your test users. After clicking Allow on the authorization page, you should see something that looks like Figure 2.1, on the next page. If you don't, make sure you've copied the code exactly. You should also make sure the test user you're using has at least one friend.

That's pretty impressive for just a few simple lines of code! Since there are only three different FBML tags in our view, let's look at each one. The first tag, `<fb:fbml>`, is similar to the `<html>` tag. It marks the start of an FBML document. Your canvas pages will still work without the `<fb:fbml>` tag, but it is required for profile pages. It's best to get into the habit of always wrapping your pages in an `<fb:fbml>` tag.

The next tag, `<fb:request-form>`, starts a special type of Facebook form. This special form allows your application to create Facebook requests.

8. http://apps.facebook.com/canvas_path/invitations/new

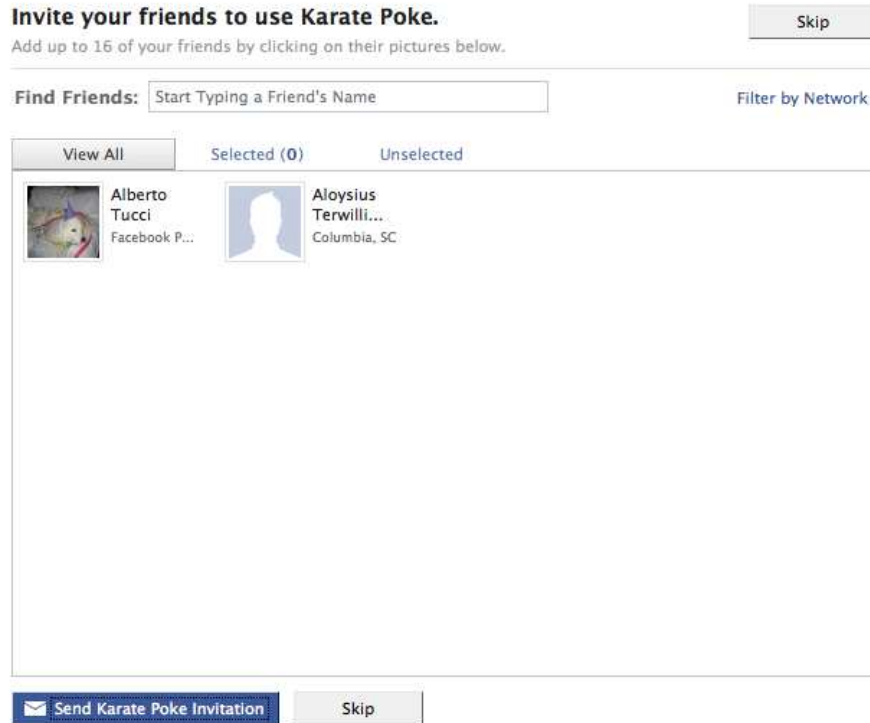


Figure 2.1: Our Facebook invitation page

It takes most of the normal form parameters such as action and method along with three additional parameters: type, invite, and content. The type parameter specifies what text shows up on the submit button. It's used to tell users what type of request they are sending. The invite parameter simply changes the title of the message. If invite is set to true, the title reads "You have a Karate Poke invitation." When set to false, it reads "You have a Karate Poke request." Finally, the content parameter gives the body of the message that is sent to the selected users.

The final tag, `<fb:multi-friend-selector>`, is responsible for rendering the actual friend selector. Facebook provides several different friend selectors. They all achieve the same goal but have different interfaces. This selector allows you to click images for your friends and includes multiple tabs. We'll look at another friend selector in Section 5.1, *Building Our First Form*, on page 86.

Working with Multiple Users

Since Facebook is a social platform, you should plan to spend a lot of time using two accounts. We'll spend a lot of time sending requests and notifications between two users. You'll get really frustrated if you have to log out of one account and log in to another every time you want to view the invitation you just sent.

To make life a little easier, I normally use two different browsers, either Firefox and Safari or Firefox and Internet Explorer depending upon my current development platform. This will allow you to stay logged in to both accounts at once. As a bonus, you can test your CSS in multiple browsers at the same time too!

If it feels like I'm going really fast, don't worry. We have only a few more FBML tags to examine. After that, we won't learn any new FBML until we get to Chapter 5, *Getting Into the Facebook Canvas*, on page 85. If you want to learn more before then, you can find a wealth of information about FBML on the Facebook wiki.⁹

So, let's try the invitation functionality. (The Skip button doesn't work yet, but it will later when we revisit invitations in Section 6.4, *Spreading by Invitation*, on page 128.) Use the friend selector to send a request to one of your test accounts. We set the action of the form to `new_invitation_path`, so we'll end up right back on this page after sending an invitation. Try that now. It's pretty hard to tell that your request was sent. Let's make a note to fix that later.

Now that we've sent a request, let's see what it looks like for the recipient. Log in as the user you sent the request to, and take a look at the upper-right corner of your home page.¹⁰ You should see a message telling you that you have a new request. When you click that message, you will be taken to a page where you can see the request. Our request is a little boring. It certainly doesn't invite you to interact with our application. It would be much nicer if the request asked you to take some action.

We have a basic invitation page working, but it's confusing for the sender and boring for the recipient. We'll make things less confusing for the sender next.

9. <http://wiki.developers.facebook.com/index.php/FBML>

10. By home page, I mean <http://www.facebook.com/home.php>.

2.3 Giving the Sender Some Feedback

Let's change our invitation form to tell the sender to whom they sent an invitation.

Earlier, I mentioned that `<fb:request-form>` takes the normal form parameters. One of those parameters, `action`, controls where the user is directed when the form is submitted. Let's change our controller to have a `create` method and send the user there.

Let's change `new.erb` to post the form to the `create` action:

```
<fb:fbml>
  <fb:request-form
    action="<%=invitations_path%>"
    method="POST"
    invite="true"
    type="Karate Poke"
    content="Attack your friends. Install Karate Poke now." >
    <fb:multi-friend-selector
      showborder="false"
      actiontext="Invite your friends to use Karate Poke." />
    </fb:request-form>
  </fb:fbml>
```

Now we're ready to implement the `create` action. As part of the invitation process, Facebook will send our application the list of recipients for our invitation. We get this list in the `ids` parameter. Let's take this list of IDs and display it to the screen:

Download `chapter2/karate_poke/app/controllers/invitations_controller.rb`

```
def create
  @sent_to_ids = params[:ids]
end
```

Now that we have a list of users, we need a view to display them. Rails is going to try to render `create.erb`, so let's create that file in `app/views/invitations`:

```
<fb:fbml>
<% for user in @sent_to_ids %>
  <%= user %> <br />
<% end %>
<%= link_to "Send another invitation",new_invitation_path %>
</fb:fbml>
```

Give it a try! After you send an invitation, you should see a list of integers on your screen. Those are the Facebook IDs of our invitation's recipients. Facebook uniquely identifies each user with an integer ID.

Showing the IDs is a little bit better than just showing the form again, but not much. Instead of showing the user IDs, let's show the users' names and profile pictures.

We could ask Facebook to give us this information (we'll learn how in Section 3.7, *Using the Facebook API*, on page 69), but instead, we'll make Facebook do the work for us. We can use two special FBML tags here: `<fb:name>` and `<fb:profile-pic>`. Both take a `uid` parameter that tells Facebook which user's information to show:

```
Download chapter2/karate_poke/app/views/invitations/create.erb
```

```
<fb:fbml>
  <% for id in @sent_to_ids%>
    <fb:profile-pic uid="<%=id%" />
    <fb:name uid="<%=id%" />
    <br />
  <% end %>
</fb:fbml>
```

OK, try it. You should see the image of the friend to whom you sent the request. In general, we want to make Facebook do as much work as possible. It's not just because we're lazy but also because it will make our pages load faster. We'll use FBML instead of the API whenever we get the chance. With that finished, let's move on to making our requests more interactive.

2.4 Making Our Invitation Interactive

We have made our invitation-sending system more interesting for the sender, but the recipient still can't do anything with it. Let's add a button to the request. We'll use yet another FBML tag, `<fb:req-choice>`, to add a Send Invitation button to our message.

`<fb:req-choice>` is a simple tag that works a lot like a link. You provide a URL and a label, and Facebook will render a button in the request. Let's add a button to our request that takes the user to our invitation form. We'll also include the ID of the sender in our request so that we can track who invites the most users.

First, we're going to need to get the Facebook ID of the sender. Let's create a new action in `invitations_controller.rb` to get this data. We can get the user ID from Facebook using the `facebook_session` object. We'll talk more about what this does in Section 2.5, *Updating the Profile*, on the following page.

```
def new
  @from_user_id = facebook_session.user.to_s
end
```

With that done, let's return to `new.erb`. I mentioned that `<fb:req-choice>` adds a button to our invitation's message. That means we need to add the tag to the content attribute of the `<fb:request-form>` tag.

Since we want to send the recipient to the new action, we can use our `new_invitation_url` as the value for the `url` attribute. Ignore the `:canvas=>true` parameter for now. We'll talk about that in more detail in Section 6.5, *Giving the Profile a Makeover*, on page 131. Also, make sure to use `new_invitation_url` and not `new_invitation_path`. You'll see why in a bit.

```
content="Attack your friends. Install Karate Poke now. \
<fb:req-choice
  url="<%=new_invitation_url(:from=>@from_user_id,:canvas=>true)%>"
  label="Attack!" />"
```

That's almost right. Since we're adding a tag inside another tag, we're going to have to do some escaping. We can use the Rails `h` method to escape our HTML:

```
content="Attack your friends. Install Karate Poke now. \
<%=h "<fb:req-choice
  url="\#{new_invitation_url(:from=>@from_user_id,:canvas=>true)}\"
  label="\ "Attack!\ " />" %>"
```

Give it a try. You should see a button at the bottom of your request. I know the view code looks a little ugly. We'll clean that up before the end of the chapter.

2.5 Updating the Profile

Now that we have invitations working, let's make it more obvious who has installed our application by putting some information in our users' profiles. Let's add a message telling the world who invited our user to install the application. Remember that a user's profile is different from a canvas page. We're going to have to ask Facebook to store some profile information for us.

Facebooker provides a very nice interface for working with the Facebook API. Through this interface we have access to objects that represent users, events, networks, and many other Facebook objects.

Access to this API starts in the controller with the `facebook_session` method, which always gives you the session object for the current

Growing Virally

You've probably heard the term *viral growth*. Viral growth is the spread of something from one person to another. It's named after the way viruses like the flu spread. One person gives it to a few of their friends, who give it to a few of their friends. Before you know it, a large part of a community is infected.

Facebook applications tend to spread this way. Facebook provides tools such as invitations, notifications, and news feed items that can help your application spread without needing a huge advertising budget. Although these tools can help your application spread, nothing is as important as creating an application that your users want to use.

Since viral growth depends on your users encouraging new users to join, we can measure the *viral coefficient* of your application by looking at the average number of new installations generated by each new user. For instance, if each new user convinces three of their friends to join, you have a viral coefficient of 3.0. If you have 100 users and they convince 130 of their friends to join, you have a viral coefficient of 1.3. If your application starts with 100 users and a viral coefficient of 1.3, it will have almost 8,000 users in twelve weeks. By week 24, your application will have more than 180,000 users.

If instead your application generated only one install for each new user, your application would have only 1,200 users after twelve weeks and 2,400 users after twenty-four weeks. Things are even worse if your application has a viral coefficient less than one. A small difference in your application's viral coefficient makes an enormous difference in how your application spreads.

There's no magic bullet for making your application viral. You can monitor your viral coefficient as you go and see how your changes affect it. Keep the changes that increase your coefficient, and eliminate those that don't. Your goal is to keep it to more than one.

Even if you manage to create an application with a high viral coefficient, don't expect it to stay high forever. At some point your application's growth will start to fall off. That happens when you saturate your market. Don't feel bad. If that didn't happen, we'd all have the flu all the time.

viewer. For instance, to get a list of photo albums for the current user, we could use `facebook_session.user.albums`. We can even view the current contents of a user's profile area using `facebook_session.user.profile_fbml`.

Some values, like `profile_fbml`, can be changed as well:

```
@user = facebook_session.user
@user.profile_fbml = "<fb:fbml>Profile Content</fb:fbml>"
```

Now that we know how to add content to a user's profile, let's add some content of our own. When a user adds our application, let's add a link to their profile that their friends can use to install Karate Poke. In Section 2.3, *Giving the Sender Some Feedback*, on page 43, we added a button to the request that took the receiver to the new invitation page and set the `from` parameter on the link. Let's add a check for that parameter in our new action and set some profile content for new users.

It can take several seconds to write to a user's profile, so we don't want to write to a user's profile each time they load the page. Every time we use the Facebook API we're sending an HTTP request to the Facebook servers.¹¹ Even just setting the profile content is enough to cause a noticeable slowdown. To minimize the slowdown caused by updating a user's profile, let's add content only if the user comes from an invitation:

```
def new
  if params[:from]
    @user = facebook_session.user
    @user.profile_fbml =
      "<fb:fbml>
      <a href='"+
      new_invitation_url(:from=>@user.to_s,:canvas=>true)+
      "'>Attack your friends, install Karate Poke</a>"+
      "<br />I was sent here by "+
      "<fb:name uid='#{params[:from]}' /></fb:fbml>"
  end
  @from_user_id = facebook_session.user.to_s
end
```

Even though our application will set profile information for our users, it won't necessarily be displayed. Facebook requires that a user grant an application permission to add information to their profile before it shows up. We'll need to add a special tag to our view to ask for that permission.

11. The Facebook session object looks like a regular Ruby object but behaves differently. This is what Joel Spolsky calls a *leaky abstraction*. See <http://www.joelonsoftware.com/articles/LeakyAbstractions.html>.

We'll need to add the `<fb:add-section-button>` tag to our `new.erb` file. This tag renders a button when our application has set profile content for a user and that user hasn't already given permission to display content in their profile. Add the following code to the beginning of our `new.erb` file:

```
<fb:add-section-button section="profile" />
...
```

Try that. After you send an invitation to one of your test accounts, log in as the test account, and look at your requests. You should see the request you just sent. Once you find that request, click the Attack! button. You will be taken to the invitation page we built. After clicking the Add to Profile button, you should also have a new box in your profile!

I'm glad that this works, but I'm uncomfortable having view code in our controller. Let's clean that up a little. Let's move our profile display to a partial and use `render_to_string` to get the content. We're using `render_to_string` instead of `render` here for a reason. `Render` sends a page back to the browser. We don't want to send the profile content to the browser; we want to turn it into a string we can send to Facebook. In the end, our view should look like this:

Download chapter2/karate_poke/app/views/invitations/_profile.erb

```
<fb:fbml>
  I was sent here by <fb:name uid='<%= from %>' />
</fb:fbml>
```

And our new action should look like this:

```
def new
  if params[:from]
    @user = facebook_session.user
    @user.profile_fbml =
      render_to_string(:partial=>"profile",
        :locals=>{:from=>params[:from]})
  end
  @from_user_id = facebook_session.user.to_s
end
```

That still feels a little ugly. We can easily make it more readable.

```
def new
  if should_update_profile?
    update_profile
  end
  @from_user_id = facebook_session.user.to_s
end
```



```

def should_update_profile?
  params[:from]
end

def update_profile
  @user = facebook_session.user
  @user.profile_fbml =
    render_to_string(:partial=>"profile",
      :locals=>{:from=>params[:from]})
end

```

The controller doesn't feel like quite the right place to be updating our users' profiles, but that's good enough for now. We'll see a better way to update profiles in Section 6.5, *Giving the Profile a Makeover*, on page 131. Let's move on to some more important cleanup.

2.6 Refactoring to Use Helpers

We've built all the functionality we talked about for this chapter, but we're not quite done yet. We shouldn't finish the chapter without doing a little cleanup. After all, building FBML by hand doesn't feel very Railsy.

It would be nice if there were helpers we could use to make our code easier to read. Thankfully, Facebooker provides us with FBML helpers. We'll use a few of them to clean up our views.

Let's start by cleaning up our invitation form. Including our message as an attribute and having to escape the FBML in it was really ugly. Facebooker provides us with the `fb_multi_friend_request` helper to replace the entire `<fb:request-form>` that we used previously. This helper uses a block to receive the message:

```

<fb:fbml>
  <fb:add-section-button section="profile" />
  <% fb_multi_friend_request("Karate Poke",
    "Invite your friends to use Karate Poke.",
    invitations_path) do %>
    Attack your friends. Install Karate Poke now.
    <%= fb_req_choice("Attack!",
      new_invitation_url(:from=>@from_user_id, :canvas=>true))%>
  <% end %>
</fb:fbml>

```

That feels much better! Compare our new version to this:

```
<fb:fbml>
  <fb:add-section-button section="profile" />
  <fb:request-form
    action="<%=new_invitation_path%"
    method="POST"
    invite="true"
    type="Karate Poke"
    content="I added a cool application." >
    <fb:multi-friend-selector
      showborder="false"
      actiontext="Invite your friends to use Karate Poke." />
  </fb:request-form>
</fb:fbml>
```

That feels much more like Rails code. The `fb_multi_friend_request` helper handles both the `<fb:request-form>` and `<fb:multi-friend-selector>` tags for us. Inside the block, we can specify request choices using the `fb_req_choice` helper.

We're going to use the Facebooker helpers throughout the rest of this book. We'll look at the generated FBML the first time we use a helper. If you ever want to see the FBML that a helper generates, you can see it by viewing the source of the page that Facebook sends to your browser. Facebook includes the FBML as a comment to developers of applications. Let's take a look at that now. Go to your invitation page as the user who created the application, and view the source in your browser:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en" id="facebook">

<!--Rendering the page using the following FBML retrieved from
http://mingus.example.com:3000/invitation You are seeing this
because you are a developer of the application and this information
may be useful to you in debugging. The FBML will not be shown to other
users visiting this page. (dashes were replaced with underscores):

<fb:fbml>
  <fb:request_form
    action="/karate_poke/invitation"
    method="POST"
    invite="true"
    type="Karate Poke"
    content="Attack your friends. Install Karate Poke now." >
```

```
<fb:multi_friend_selector
  showborder="false"
  actiontext="Invite your friends to use Karate Poke." />
</fb:request_form>
</fb:fbml>
-->
```

Unfortunately, as the comment says, Facebook writes the FBML into pages only when you are the developer of the application. Since test accounts can't be listed as developers, that means we can't view the FBML using our test accounts. Even with that restriction, the ability to view generated FBML can make it much easier to debug FBML issues.

Now that we've cleaned up the invitation, let's clean up our create view and our profile view. We built `<fb:name>` and `<fb:profile-pic>` tags by hand in both of those. We can use the `fb_name` and `fb_profile_pic` helper methods instead:

```
<fb:fbml>
  <% for id in @sent_to_ids%>
    <%= fb_profile_pic id %> <%= fb_name id %><br />
  <% end %>
</fb:fbml>
```

OK, just one more cleanup to go. Since we're typing `<fb:fbml>` in every view, let's move that to our `app/views/layouts/application.erb` file:

```
<fb:fbml>
  <%= yield %>
</fb:fbml>
```

I feel much better after doing that cleanup. I hope you do too.

2.7 Summary

Wow, this has been a productive chapter. We created a new Rails application and turned it into a Facebook application. We built an invitation interface and updated our users' profiles. Along the way, we even got an introduction to FBML. That was a lot to learn, but don't worry—we'll slow down a little when we build the Karate Poke object model.

Chapter 3

Building the Karate Poke Object Model

We've started building Karate Poke, but we can't do very much with it yet. In fact, all we've done is send an invitation. In this chapter, we'll build the object model of Karate Poke. We'll start by creating a User model. With that in place, we will create a Move model and use it in an Attack model. With the basics done, we'll add back-end support for some more advanced social features. We'll wrap up with a look at common causes of performance problems caused by the model layer.

Even though we're building an object model for Karate Poke, you'll see a lot of concepts that can be used in other applications. We'll look at how to create a model to represent Facebook users. We'll see how to access Facebook sessions from inside our models. Finally, we'll look at some performance topics that are particularly important to Facebook applications.

3.1 Building the User Model

It seems like every Rails application has a User model in it. Karate Poke is no exception. Our User model will be different from the ones we've built before. For instance, we won't need to worry about authentication, since Facebook already handles that for us. We also won't need to write code to handle user sign-up. Once again, Facebook has it covered. We won't even need to store much information about the user, since we have access to the Facebooker::User class, which we'll see in more detail in Section 3.7, *Using the Facebook API*, on page 69.

Our User model will primarily act as a bridge between the Facebook::User class and our other models. We'll start by building a simple User model with just a few fields. Next, we'll build methods for creating users. With that in place, we can integrate our model with our controllers.

Creating the User Model

Since our User model doesn't need to store much data, it should be really easy to build. Let's start by creating the model:

```
$ script/generate model User
```

Now we just need to edit the generated migration. We'll need only two fields and the timestamps:

```
Download chapter3/karate_poke/db/migrate/002_create_users.rb
```

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.integer :facebook_id, :limit=>20, :null=>false
      t.string :session_key
      t.timestamps
    end
  end

  def self.down
    drop_table :users
  end
end
```

The `facebook_id` field should look familiar. We saw it in Section 2.3, *Giving the Sender Some Feedback*, on page 43. You can ignore the `session_key` field for now. We won't need to worry about it until Section 3.2, *Accessing Facebook from Models*, on page 57.

Notice that there is no name field. There's a good reason we didn't include one. Facebook's terms of service limit what information we can store in our application. We'll talk about this in Section 3.1, *What Information Can We Store?*, on page 56.

Before we can use our User model, we'll need to run `rake db:migrate`. Our application will also need to create User instances. Let's do that next.

Putting Our User to Work

Now that we have a User model, let's use it in our application. We'll need at least two points of integration. First, we'll need to create a User

Facebook and 64-Bit IDs

Our user migration included something you don't often see. By specifying a limit of 20 on our `facebook_id` column, we asked Rails to create a 64-bit integer. Facebook recently announced a move to 64-bit user IDs. I guess it expects to have more than 4 billion users soon.

That's good news if you're building an application that will be used by every human on the planet, but it can cause a little pain for MySQL users. Although most databases support 64-bit integers, MySQL migrations didn't until Rails 2.1. If you're on Rails 2.1, you should be ready to go. If you're on an earlier version of Rails, you'll need to install the MySQL bigint plug-in.*

*. Available at http://svn.northpub.com/plugins/mysql_bigint

instance whenever a user accesses our application for the first time. Second, we'll need a way to access the User model for the current viewer.

We talked about how Facebook uses an ID parameter to uniquely identify each user in Section 2.3, *Giving the Sender Some Feedback*, on page 43. Let's add a method that will create a User instance given a Facebook ID.¹

```
class User < ActiveRecord::Base
  def self.for(facebook_id)
    User.create_by_facebook_id(facebook_id)
  end
end
```

Now we need to figure out where to use this method. Our first goal is to make sure we have User instances for each of our users. We can do that by creating a before filter in `application.rb`:

```
before_filter :create_user

def create_user
  User.for(facebook_session.user.to_i)
end
```

1. You won't be able to find the `create_by_facebook_id` method in any of the Rails or Facebooker documentation. It is a dynamic method that is created by ActiveRecord on first use. See <http://api.rubyonrails.com/classes/ActiveRecord/Base.html> for more information.

Our filter simply calls our `for` method and passes in the viewer's Facebook ID. We used the `facebook_session` briefly in the previous chapter. We'll look at it in more detail in Section 3.2, *Accessing Facebook from Models*, on page 57.

Oops, I see a problem. We're going to end up creating a lot of `User` objects. Let's change our method to create a `User` object only if one doesn't already exist:

```
def self.for(facebook_id)
  User.find_or_create_by_facebook_id(facebook_id)
end
```

That should work better. Now, we'll have a `User` object for each person who uses our application. That takes care of our first point of integration. We also wanted a way to access the `User` instance for the current viewer. Let's follow the convention of Rick Olson's *Restful Authentication plug-in*² and use a method called `current_user` to represent the currently logged in user. We can create an attribute for the current user and set it in our filter:

```
attr_accessor :current_user
before_filter :create_user

def create_user
  self.current_user = User.for(facebook_session.user.to_i)
end
```

That works, but our filter is incorrectly named now. Let's rename it to be a little clearer:

```
attr_accessor :current_user
before_filter :set_current_user

def set_current_user
  self.current_user = User.for(facebook_session.user.to_i)
end
```

There's just one more thing we need to do. We wanted to make the `current_user` available to our views as well. We could add a `current_user` method in `ApplicationHelper`, but there's an even easier way to do this. Rails provides the `helper_attr` method for making controller methods available to views. Let's use it here:

```
Download chapter3/karate_poke/app/controllers/application.rb
```

```
helper_attr :current_user
```

2. Described at <http://weblog.techno-weenie.net/2006/8/1/restful-authentication-plugin>

That takes care of everything we need for user management. It feels good not to have to worry about email address validation and all the hassles of user creation. Let's forge ahead with our object model.

What Information Can We Store?

Before you store information about your Facebook users, make sure you take a look at the Facebook terms of service.³ Facebook allows you to store only the IDs of events, networks, and users. Although that sounds extremely limiting, it isn't that bad. In fact, Facebook has two good reasons for this policy.

First, by requiring your application to request information on each page view, it can provide a very fine-grained privacy implementation. Each request you send to Facebook, either as an FBML tag like `<fb:name>` or via an API request, is executed in the context of the current viewer. For example, if only friends can see my name, `<fb:name>` will return "Mike Mangino" for my friends and "" for people who aren't my friend. The same thing happens if I call the name method on a `Facebooker::User` instance. Facebook would have a hard time enforcing this restriction if each application stored the names of its users.

Second, along with enforcing privacy controls, Facebook's policy also improves data consistency. You can think of it like a normalized database schema. Facebook is ensuring that a user's friend list is stored in only one place. If this weren't the case, each application would need to be updated every time one of their users changed any of their information. This would quickly become unwieldy.

In practice, this policy doesn't hurt as much as it sounds like it could. You'll find that there is very little information you want to store that can't be written with an FBML tag. Additionally, Facebook does relax its data storage prohibition slightly in the interest of performance. You are allowed to store information for caching purposes for up to twenty-four hours. We'll talk more about caching for performance in Section 9.2, *Caching Our Views*, on page 173, but I've never had to take advantage of this provision.

3. <http://www.facebook.com/developers/tos.php>

3.2 Accessing Facebook from Models

We've seen the `facebook_session` method several times now. The `Facebooker::Session` object represents a user's session with Facebook. Facebook uses sessions to verify that our application is performing actions on behalf of an active user. When we want to send a notification on behalf of our user, we will provide Facebook with that user's session information. Facebook will verify that the session does in fact belong to the requesting user and will also verify that the user has been active on our application within the past hour. Facebook uses the session to prevent applications from taking action on behalf of a user who isn't actively using them.

The `facebook_session` is created by `Facebooker` from our application's API key, our secret key, and the session key of the current user. The session key is a value Facebook sends our application on each request. `Facebooker` will automatically create a session object for us on each web request. If we want to use the session outside a request, we'll have a little problem. We'll see some of these situations in Section 9.4, *Move API Calls Out of Line*, on page 184.

We created a `session_key` column in our `users` table earlier. We're going to put it to use now. Let's update our `for` method to store the session key:

```
def self.for(facebook_id, facebook_session=nil)
  returning find_or_create_by_facebook_id(facebook_id) do |user|
    unless facebook_session.nil?
      user.update_attribute(:session_key,
        facebook_session.session_key)
    end
  end
end
```

You'll notice that the `facebook_session` parameter is optional. You'll see why in Section 5.1, *Implementing the Attack*, on page 86. We also used the Rails `returning` method. `returning` takes a single parameter that it will both yield and return. It is often used when you want to create a new object and call some methods on it and then return the newly created object. In our case, we call `update_attribute` to set the session.

It seems like we're updating the session key a lot. Session keys will change only when a user stops using our application for an hour or more. Since session keys don't change often, let's change our code to update the session key only when it has changed.

We can add a `store_session` method to encapsulate this logic:

Download chapter3/karate_poke/app/models/user.rb

```
def store_session(session_key)
  if self.session_key != session_key
    update_attribute(:session_key, session_key)
  end
end
```

Now we can change our `for` method to use our new `store_session` method:

Download chapter3/karate_poke/app/models/user.rb

```
def self.for(facebook_id, facebook_session=nil)
  returning find_or_create_by_facebook_id(facebook_id) do |user|
    unless facebook_session.nil?
      user.store_session(facebook_session.session_key)
    end
  end
end
```

That takes care of part of the problem. We now have access to the session key from inside our models. What we really wanted was the `Facebooker::Session` object. Since we have all the necessary fields, we can create just one:

```
Line 1 # Re-create a Facebooker::Session
- # object outside a request
- def facebook_session
-   @facebook_session ||=
5     returning Facebooker::Session.create do |session|
-       # Facebook sessions are good for only one hour after storing
-       session.secure_with!(session_key, facebook_id, 1.hour.from_now)
-     end
- end
```

There's a lot going on in such a small method. Let's break it down line by line. In line 4, we're using the Ruby conditional assignment operator. The conditional assignment operator is often used for caching. If the variable on the left side of the statement is `nil`, it runs the right side of the expression and sets the variable. If the variable is already set, it does nothing. The first time we call `facebook_session`, it will create a session and assign it to `@facebook_session`. The next time it will use the existing value.

In line 5, we use the `returning` method that we talked about earlier. Next, we call `secure_with!` on line 7. `secure_with!` is used to associate a session key with a `Facebooker::Session` object.



Figure 3.1: Moves include both a name and an image.

I told you that was a complicated method. Now that we've written it, we can access the Facebook sessions of our users. That means we can make Facebook API calls from inside our models. Before we do that, we'll need to visit our application in a web browser to set our session key. After doing that, let's move on to the rest of our model.

3.3 Creating the Move Model

Now we're starting to make some progress. With our User model in place, let's create a model to represent the different attacks our users can perform. Each Move instance will represent a karate move, such as "karate chop" or "crane kick." Let's also include an image to make our application a little more fun. You can see how our moves will be used in Figure 3.1.

Since this is such a simple model, we can create it from the command line and not even have to modify the generated migration:

```
$ script/generate model Move name:string image_name:string
```

You should run `rake db:migrate` to create the table after running the previous command. I've already created some images for us to use. You can copy them from `chapter3/karate_poke/public/images` into the `public/images` directory of your application.

Now that we have a Move model, let's allow a user to attack another user. The User model seems like a good place to put this method. It could look something like this:

```
def attack(other_user,move)
  # what goes here?
end
```

You know, I think we're still missing something. We need a model to keep track of attacks. Let's build that before we implement the attack method.

3.4 Attack!

It's time to get to the real meat of Karate Poke. Now that we have users and moves, we have everything we need to attack. Even though we're building a very specific feature for Karate Poke, the pattern we follow covers a broad range of problems. For instance, our attack code will be almost identical to the code used for sending a gift to a friend or even a message model. As you follow along, think about how you might be able to change the code to solve a different problem.

Attack Basics

Before we can create our Attack model, we need to figure out what we're building. We're not in a Jackie Chan movie, so let's assume that an attack happens between two users. We'll call them the *attacker* and the *defender*. Let's also assume that the attacker can use only one move per attack. We could get more complicated, but sometimes simpler is better. Let's create our model from the command line again:

```
$ script/generate model Attack \  
  attacking_user_id:integer \  
  defending_user_id:integer \  
  move_id:integer
```

Take a look at the migration that was generated for us:

```
Download chapter3/karate_poke/db/migrate/004_create_attacks.rb  
  
class CreateAttacks < ActiveRecord::Migration  
  def self.up  
    create_table :attacks do |t|  
      t.integer :attacking_user_id  
      t.integer :defending_user_id  
      t.integer :move_id  
  
      t.timestamps  
    end  
  end  
  
  def self.down  
    drop_table :attacks  
  end  
end
```

Along with the fields we specified, Rails added the timestamp fields. That's a good thing since we'll want to use `created_at` to show the order of attacks.

After running that migration, we can set up the associations for our `Attack` model. We'll need an association for the attacking user as well as one for the defending user. Remember to include both the `class_name` and `foreign_key` parameters. We'll also need an association between the attack and the move:

Download chapter3/karate_poke/app/models/attack.rb

```
belongs_to :attacking_user,
           :class_name=>"User",
           :foreign_key=>:attacking_user_id
belongs_to :defending_user,
           :class_name=>"User",
           :foreign_key=>:defending_user_id
belongs_to :move
```

That takes care of the `Attack` associations. Let's do the same thing with our `User` model. We'll need to create associations for both attacks and defenses:

Download chapter3/karate_poke/app/models/user.rb

```
has_many :attacks, :foreign_key=>:attacking_user_id
has_many :defenses, :class_name=>"Attack",
                  :foreign_key=>:defending_user_id
```

Great! Now we can fill in the details of our attack method:

```
def attack(other_user,move)
  attacks.create!(:defending_user=>other_user,:move=>move)
end
```

That was pretty easy. You can now attack your friends. In fact, we have gone a long time without using our application. Let's fire up `script/console` and play around with it a little:

```
>> mike = User.for(1)
=> <User id: 5, facebook_id: 1 ...
>> jen = User.for(2)
=> <User id: 6, facebook_id: 2 ...
>> move = Move.create(:name=>"karate chop")
=> <Move id: 1, name: "karate chop" ...
>> mike.attack(jen,move)
=> <Attack id: 14, attacking_user_id: 5, ...
```

That's all it takes to attack! I was able to create two users using the `for` method. After creating the users, I created a `Move` object. Finally, I attacked! Try it yourself.

Adding Battle History

Now that we can attack our friends, we'll want to show our battle history. Our battle history will be a list of all our attacks and defenses. Although we could build that by combining the attacks and defenses relationships, let's instead add a `battles` method. (This will make life easier for us a little later when we introduce pagination in Section 5.5, *Adding Pagination*, on page 101.)

`Download` chapter3/karate_poke/app/models/user.rb

```
def battles
  Attack.find(:all,
    :conditions=>
      ["attacking_user_id=? or defending_user_id=?",
        self.id,self.id],
    :include=>[:attacking_user,:defending_user,:move],
    :order=>"attacks.created_at desc")
end
```

We're really cutting up now, but I can't help feeling like something is missing.

Adding Misses

It's a little boring if every attack results in a hit. Let's change our `Attack` model to add the occasional miss.

If some of our attacks are going to miss, we'll need to store that fact somewhere. Let's add a boolean `hit` column to our `Attack` model. I'll wait while you create that migration.

Now that we have a `hit` column, let's figure out how we're going to get it populated. Let's keep it simple and just randomly decide whether an attack results in a hit. The easiest implementation is to use a `before_create` filter. Let's set one up:

`Download` chapter3/karate_poke/app/models/attack.rb

```
before_create :determine_hit

def determine_hit
  returning true do
    # make it a hit 50% of the time
    self.hit = (rand(2) == 0)
  end
end
```

Let's try our attacks in script/console again. Some of your attacks should now be misses. But before you spend too much time practicing your attacks, let's move on.

3.5 Creating the Belt Model

Many of the most popular Facebook applications include features to reward power users. Typically, these rewards take the form of an unlockable item. We'll use karate belts for this. The more a user uses our application, the more moves they will be able to access.

Creating the Belt Model

In Karate Poke, each user will have a belt. They start with a white belt and progress until they become a black belt. Each time they earn a new belt, they will gain access to new moves.

We could code this in a number of ways. Let's try a really simple implementation. Let's give each move a difficulty value. For instance, a karate chop is easy, so it could be difficulty level 1. A crane kick is tough (have you seen *The Karate Kid*?), so it's probably a level 4 or 5. Each belt will then have a difficulty level associated with it. People with white belts, which have a level of 1, can perform only those moves with a difficulty level of 1. A black belt, which has a level of 9, can perform any move with a difficulty level of 9 or less.

With that decided, let's figure out what other attributes our Belt model will need. We will definitely need a name for the belt. We might as well include a `next_belt_id` attribute to make it easy to see the progression of belts. Let's throw in a `minimum_hits` column as well. We'll talk about that one more later. That should be all we need to create our model:

```
$ script/generate model Belt name:string \  
  level:integer next_belt_id:integer minimum_hits:integer
```

There are a couple of bits of housekeeping to take care of before you run `rake db:migrate`. We need to modify the users table to include a `belt_id` column. We'll also need to add a `difficulty_level` column to the Move model:

```
Download chapter3/karate_poke/db/migrate/006_create_belts.rb
```

```
class CreateBelts < ActiveRecord::Migration  
  def self.up  
    create_table :belts do |t|  
      t.string :name  
      t.integer :level  
    end  
  end  
end
```

```

    t.integer :next_belt_id
    t.integer :minimum_hits

    t.timestamps
    add_column "users", "belt_id", :integer
    add_column "moves", "difficulty_level", :integer
  end
end

def self.down
  drop_table :belts
  remove_column "users", "belt_id"
  remove_column "moves", "difficulty_level"
end
end

```

After running `rake db:migrate`, we can set up our associations. We've heard several times that each user will have a belt, so the first association is easy:

[Download](#) chapter3/karate_poke/app/models/user.rb

```
belongs_to :belt
```

We've also talked about using `next_belt_id` to show the belt progression. Let's add that here:

[Download](#) chapter3/karate_poke/app/models/belt.rb

```
belongs_to :next_belt, :class_name=>"Belt", :foreign_key=>:next_belt_id
```

With the model set up, we will need to create some data. Luckily for you, I have already created it. You can copy the fixture files from `chapter3/test/fixtures/` to your `test/fixtures` directory. You can run `rake db:fixtures:load` to populate your development database with some basic belts and moves.

Integrating Our Belts

We have a Belt model, some moves, and some belts. It's time to integrate belts into the rest of the system. I mentioned earlier that each user starts out with a white belt. This seems like a good time to make that happen.

We can make our users start with a white belt in a few ways. We could use a default value in the database for this, but it seems ugly to put application logic there. Instead, let's use a `before_create` filter. Before we can set the belt, we'll need to know which belt is the initial belt.

Let's add an `initial_belt` method to the belt class:

Download chapter3/karate_poke/app/models/belt.rb

```
def self.initial_belt
  find_by_level(1)
end
```

Now we can set the belt in a before filter on our User model:

Download chapter3/karate_poke/app/models/user.rb

```
before_create :set_initial_belt
```

```
def set_initial_belt
  self.belt = Belt.initial_belt
end
```

Now that we have belts and levels and moves (oh my!), let's see what else needs to change. Since a user can no longer perform all the moves, we need a way to determine what moves are available to them. Let's create a method on the User model to do just this. Since we went with a really simple implementation of belts, this should be easy:

Download chapter3/karate_poke/app/models/user.rb

```
def available_moves
  Move.find(:all,
    :conditions=>["difficulty_level <= ?",belt.level],
    :order=>"name asc")
end
```

Great! There's just one thing we talked about but haven't implemented. We need a way for our users to earn new belts. This is where the `minimum_hits` column we added earlier comes into play. We will upgrade a user to a new belt once their successful attack count reaches the next belt's `minimum_hits` value.

Since we're going to base our upgrades on the total number of hits a user has, we should add a method to make it easy to find that bit of information:

```
def total_hits
  attacks.count(:conditions=>{:hit=>true})
end
```

Now we can add a method to our Belt model to determine whether a user should be upgraded:

Download chapter3/karate_poke/app/models/belt.rb

```
def should_be_upgraded?(user)
  !next_belt.nil? and user.total_hits >= next_belt.minimum_hits
end
```

There are two conditions necessary to upgrade a user. First, there must be a higher belt. If `next_belt` is nil, we know that the user has the highest belt. If there is a higher belt, we check the `minimum_hits` parameter to see whether the user qualifies for it.

Since we upgrade only after a successful attack, let's put that code into the attack method:

```
def attack(other_user,move)
  returning attacks.create!(:defending_user=>other_user,:move=>move) do
    if belt.should_be_upgraded?(self)
      update_attribute(:belt,belt.next_belt)
    end
  end
end
```

There's one last step before we can call our Belt model done. We need to update all the existing users in our database to make sure they have a belt. We can do that from script/console:

```
User.find(:all).each do |user|
  user.update_attribute(:belt,Belt.initial_belt)
end
```

Wow, we had to make changes all over the place to implement our belt system. We have one more feature to add before we spend some time cleaning up.

3.6 Encouraging Invitations

Along with rewarding power users, successful Facebook applications often reward users who help their application spread. Let's add a way to track which users send the most invitations. We'll create a leaderboard to show this off in Chapter 8, *Integrating Your App with Other Websites*, on page 158.

In Section 2.4, *Making Our Invitation Interactive*, on page 44, we added an entry to a user's profile that told who referred them to our application. We're going to adapt that a little to develop a dojo (literally "place of the way," which is a formal gathering place for training in martial arts) system. We'll associate each invited user with the sensei (a Japanese title used to refer to authority figures that is often used in martial arts to refer to a teacher) who invited them. As a user invites more people, they will increase the number of disciples in their dojo.

This will be really easy to implement in our models. In fact, we can do it with just a single column and a couple of methods. Let's start by adding

a `sensei_id` column to our `User` model. Go ahead and create a migration for this.

Now that we have a `sensei ID`, let's add some relationships. We can add the `sensei` relationship as well as the `disciple` relationship:

Download `chapter3/karate_poke/app/models/user.rb`

```
belongs_to :sensei, :class_name=>"User", :foreign_key=>:sensei_id
has_many :disciples, :class_name=>"User", :foreign_key=>:sensei_id
```

It feels a little strange to have two relationships in the same model using the same column. Even so, it does exactly what we need.

Since we want to encourage our users to spread our application, let's add a way to determine which of a user's friends have already added the application:

Download `chapter3/karate_poke/app/models/user.rb`

```
def friends_with_senseis(friends_facebook_ids)
  User.find(:all,
    :conditions=>["facebook_id in (?) and sensei_id is not null",
      friends_facebook_ids])
end
```

That's all we need to do on the model side. We'll talk more about dojos when we discuss invitations in Section 6.4, *Spreading by Invitation*, on page 128. Next we'll take a look at how we can get data out of Facebook.

3.7 Getting Data Out of Facebook

Now that we have a basic data model for our application, let's take a look at the Facebook data model. Facebook provides a REST API that we can use to obtain data about our users. We talked a little about REST in Section 2.2, *Creating the Invitations Controller*, on page 39. We'll start by looking at what happens each time we make a request. Next, we'll walk through some of the API methods that are available to us.

How the REST API Works

To really understand how the REST API works, we'll step through an example. In our example, we will use `script/console` to retrieve our name from Facebook. Start `script/console`, and find your `User` instance. Once you have your `User` object, run the following code:

```
>> user.facebook_session.user.name
=> "Mike Mangino"
```

That doesn't look like anything special because Facebooker encapsulates the Facebook API behind a very Ruby-like façade. Behind the scenes, Facebooker does a lot of work to retrieve our name. First, Facebooker sends a POST to the Facebook API service. It includes a number of parameters in the request such as the `api_key` of our application, the `session_key` of the user making the request, and the `uid` of the user whose name we want to retrieve. Additionally, Facebooker adds the `fb_sig` parameter as proof that our application is making the request. We talked about signatures earlier in Section 2.1, *The Details of Facebook Signatures*, on page 37. By requiring all API calls to be signed, Facebook can verify that requests are coming from an approved application.

In response to our request, Facebook will return an XML document similar to the one shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<users_getInfo_response xmlns="http://api.facebook.com/1.0/" ...>
  <user>
    <uid>12451752</uid>
    <status>
      <message/>
      <time>0</time>
    </status>
    <political/>
    <pic_small>http://profile.ak.facebook.com/profile...</pic_small>
    <name>Mike Mangino</name>
    <quotes/>
    <is_app_user>1</is_app_user>
    <tv/>
    <profile_update_time>0</profile_update_time>
    <meeting_sex list="true"/>
    <hs_info>
      <hs1_name>Westerville - North High School</hs1_name>
      <hs2_name/>
      <grad_year>1996</grad_year>
      <hs1_id>19941</hs1_id>
      <hs2_id>0</hs2_id>
    </hs_info>
    <timezone>-6</timezone>
    <relationship_status>Married</relationship_status>
    ...
```

When Facebooker receives the response, it turns the XML into Ruby objects. Each time Facebooker needs to load more information, it sends an HTTP request to Facebook. Each request takes some time, typically between a quarter and a half of a second. We'll want to keep this timing in mind as we use the REST API.

Explore the Available Data

One of the biggest benefits of Facebook development is the amount of data to which you have access. Make sure you explore this data. For example, Growing Gifts displays a list of your friends' upcoming birthdays so that you can send them a birthday flower. This type of integration would be next to impossible without Facebook.

You're not limited to just birthdays. In Karate Poke, we could look at the education history of your friends and suggest you attack your friends who went to rival schools. For other applications, maybe they could use the list of groups a user is a member of or information about upcoming events they will be attending. As you build your application, think about what you can do with the data that Facebook provides!

Using the Facebook API

We've looked at several uses of the Facebook API already. Since Facebooker makes the Facebook API look just like regular Ruby objects, we aren't going to spend time looking at every object.⁴ Instead, we'll look at a typical use of the API.

For our example, we'll use the REST API to add a method to our User model that will display a user's hometown. Our hometown method will need to access the hometown_location attribute on the Facebooker::User object. For that, we'll need to create an instance of Facebooker::User to represent the user in question. To create a user, we can simply call the new method and pass in the Facebook ID of the user we want. After creating the user instance, we retrieve the hometown_location. If the location is blank, we provide default text.

```
def hometown
  fb_user = Facebooker::User.new(facebook_id)
  location = fb_user.hometown_location
  text_location = "#{location.city} #{location.state}"
  text_location.blank? ? "an undisclosed location" : text_location
end
```

Let's give that a try in script/console. Before we do, we need to understand how Facebook fetches data. Each instance of Facebooker::User can

4. You can find the Facebook REST API documentation online at <http://wiki.developers.facebook.com/index.php/API>.

hold a reference to a session object. When we access a user through the session, by calling `User.find(1).facebook_session.user`, for example, Facebooker will link that session and the user object. Any time the `Facebooker::User` needs to get data from Facebook, it will use the associated session. Facebook uses the session to determine the context of a request. For example, if my session ID is used to request the hometown location of somebody who isn't my friend, Facebook may not return any information.

If we create a `Facebooker::Session` object using the new method, like we did earlier, Facebooker doesn't have a session to use for fetching data. By default, it tries to use one stored in `Facebooker::Session.current`. When Facebooker receives a request through the Web, it will automatically set `Facebooker::Session.current` to the session belonging to the user making the request. During testing, we set the current session to the session for our user. Let's see an example of using our new method:

```
>> u=User.for(12451752)
=> #<User id: 5>
>> Facebooker::Session.current=u.facebook_session
=> #<Facebooker::Session:0x22ab0a0>
>> u.hometown
=> "Westerville Ohio"
```

With that in place, we have a way to get the hometown of a user. Along the way, we've seen how the Facebook REST API works. Next, we'll do a little refactoring.

3.8 Refactoring and Performance

We made good progress, but we can improve on a couple of things. We did a good job of keeping our code looking good, but we didn't really think about performance. Many of the biggest performance problems in Facebook applications are caused by poorly coded models. We'll look at some simple changes we can make to avoid these common problems.

Although performance is important to all applications, it is especially important to Facebook applications for a couple of reasons. Facebook enforces a very short timeout window. If your application does not respond to Facebook within eight seconds, your user will see an error like the one shown in Figure 3.2, on the following page. If that isn't bad enough, you also should be prepared to handle large amounts of traffic quickly. I've worked on several applications that reached more than 100,000 users in two days. Needless to say, I wish I had paid a little more attention to performance up front.

Error while loading page from karate_poke

The URL `http://cab.elevatedrails.com:3002/` did not respond.

There are still a few kinks Facebook and the makers of `karate_poke` are trying to iron out. We appreciate your patience as we try to fix these issues. Your problem has been logged – if it persists, please come back in a few days. Thanks!

Try Again

Go Home

Figure 3.2: You'll see this error if your application doesn't respond within eight seconds.

Adding Indexes

Let's start by adding a few indexes. As with any Rails application, we want to add indexes on our foreign keys. Let's create a migration to add them to our `attacks` and `users` tables. We'll need to create two indexes on `attacks` since we access it both by the `attacking_user_id` and the `defending_user_id`. We can include the `created_at` column in the indexes to help with sort performance.

Download `chapter3/karate_poke/db/migrate/008_add_indexes.rb`

```
class AddIndexes < ActiveRecord::Migration
  def self.up
    add_index :attacks, [:attacking_user_id, :created_at]
    add_index :attacks, [:defending_user_id, :created_at]
    add_index :users, :facebook_id
  end

  def self.down
    remove_index :attacks, [:attacking_user_id, :created_at]
    remove_index :attacks, [:defending_user_id, :created_at]
    remove_index :users, :facebook_id
  end
end
```

We don't need to add indexes to the `moves` and `belts` tables. It's unlikely that we would see any performance benefit from indexing them. We need to add indexes only to large tables or tables that grow over time.

Removing Queries

Although it's good to make our queries faster, it's even better to just eliminate them altogether. Let's look at a couple of changes we can make to remove a few queries from our application.

When we implemented our Belt model, we used a `total_hits` method that counted the number of times a user has hit another user. Instead of running a SQL query every time, let's just save that count in a column on the users table. Start by adding a `total_hits` column to the User model. After you create and run that migration, we can modify our `attack` method to increment the count after each hit. We'll also need to remove our existing `total_hits` method.

Download `chapter3/karate_poke/app/models/user.rb`

```
def attack(other_user,move)
  returning attacks.create!(:defending_user=>other_user,
                             :move=>move) do |a|
    if a.hit?
      increment :total_hits
      if belt.should_be_upgraded?(self)
        self.belt=belt.next_belt
      end
    end
    save!
  end
end
```

That takes care of one extra query. Let's look at some of the remaining queries that get run. We know we are going to display a list of battles for each user on our main page. Let's take a look at our `log/development.log` to see what Rails does when we call our `battles` method:

```
Attack Load (0.000557)  SELECT * FROM attacks WHERE (...
Attack Columns (0.002195)  SHOW FIELDS FROM attacks
User Load (0.001515)  SELECT * FROM users WHERE (users.'id' = 4)
User Load (0.000297)  SELECT * FROM users WHERE (users.'id' = 3)
Move Columns (0.003177)  SHOW FIELDS FROM moves
Move Load (0.000560)  SELECT * FROM moves WHERE (moves.'id' = 2)
User Load (0.000450)  SELECT * FROM users WHERE (users.'id' = 4)
User Load (0.000263)  SELECT * FROM users WHERE (users.'id' = 3)
Move Load (0.000262)  SELECT * FROM moves WHERE (moves.'id' = 2)
User Load (0.000253)  SELECT * FROM users WHERE (users.'id' = 4)
User Load (0.000249)  SELECT * FROM users WHERE (users.'id' = 3)
```

Wow, that's a lot of queries. When our `battles` method loads the list of attacks, it isn't fetching the associated User and Belt objects at the same time. We can use the `:include` option on our call to the `find` method to

Isn't Premature Optimization Bad?

Premature optimization is definitely a bad thing. Donald Knuth said this in *Structured Programming with go to Statements* (Knu74): "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

In this case, however, we're not talking about micro-optimization. We're really eliminating major bottlenecks. You could put this type of cleanup off until the end of writing the application, but I prefer to do it right away. After all, Facebook applications can get a lot of traffic quickly. It's best to at least pick the low-hanging fruit.

Additionally, this isn't an exhaustive list of performance changes. I'll talk about more Facebook-specific performance enhancements in Chapter 9, *Scaling and Performance*, on page 170.

combine all this into one query. Let's make sure our `battles` method includes the `attacking_user`, the `defending_user`, and the `move`:

Download `chapter3/karate_poke/app/models/user.rb`

```
def battles
  Attack.find(:all,
    :conditions=>
      ["attacking_user_id=? or defending_user_id=?",
        self.id, self.id],
    :include=>[:attacking_user, :defending_user, :move],
    :order=>"attacks.created_at desc")
end
```

Finally, when we load a user in our `for` method, we should use `:include` to load their belt and the next belt. Go ahead and make that change now.

That's enough performance tuning for now. That takes care of the major bottlenecks in our application. We were able to quickly add indexes to tables that will be repeatedly accessed. We also were able to eliminate a large number of database queries by strategic usage of the `:include` option.

3.9 Summary

We now have an object model that will let us build the rest of Karate Poke. We built a User model and used it to save a Facebook session. We then built all the infrastructure necessary for one user to attack another. Finally, we added belts and dojos to encourage interactivity.

That takes care of the majority of our object model. We'll add a little bit more as we go, but it's nothing substantial. We've made good progress, but there's something we've been neglecting. With all the code we've written, we still don't have a single test. We'll take care of that next.

Chapter 4

Testing Our Facebook Application

Now that we have a better understanding of how to develop Facebook applications, let's turn our attention to testing. Writing tests for our Facebook application will feel quite familiar. Instead of spending time on what you've seen before, we'll focus our efforts on what makes testing a Facebook application different from other Rails applications. If you don't have much experience writing tests with Rails, check out the excellent *Agile Web Development with Rails* [TH05].

We'll start our look at testing by focusing on controller tests. We will look at the Facebook-specific helpers we can use to make writing tests easier. From there, we'll turn our attention to model and publisher tests. We will see how to use mocks and stubs to isolate our code from Facebook.

4.1 Controller Tests

If you've tried to write Rails functional tests for your Facebook code, you've probably noticed that your tests don't work. Even a simple test like the one shown here fails. Instead of rendering the `new.fbml.erb` template, we are redirected to the application authorization page.

```
def test_new
  get :new
  assert_response :success
  assert_template 'new'
end
```

We talked about Facebook signatures earlier in Section 2.1, *The Details of Facebook Signatures*, on page 37. In our previous test, we didn't send the right Facebook signature parameters. That makes Facebooker redirect to the application authorization page. Although this is the behavior we want for our application, it makes testing more difficult.

To make testing easier, we'll want to isolate ourselves from Facebook. We'll look at two different ways to do this. We'll start by using the `facebook_post` method to automatically create the correct signature for our application. We'll also look at using *FlexMock*, a mock object framework from Jim Weirich, to isolate ourselves from the Facebook API. Along the way, we'll create tests for our most important actions.

Before we can start testing, we have a little setup to do. First, we'll need to make sure our `facebooker.yml` is configured for the test environment. Since our application won't talk to Facebook in test mode, we don't need to create a new Facebook application for testing. We can just use our existing development configuration information.

Along with configuring our Facebook setup, we're also going to need to create some basic data for testing purposes. I've already created basic fixture data you can use. Just copy the files from `chapter4/karate_poke/test/fixtures` file into your `fixtures` directory.

Testing with the Facebooker Helpers

For our first functional test, we're going to start simple. Let's make sure we can view our invitation form. We'll check to make sure the new template is rendered. A first version of our test might look like this:

```
def test_new_invitation
  get :new
  assert_response :success
  assert_template 'new'
end
```

It's a good first version, but we're missing a few things. As mentioned earlier, we need to pass in the Facebook signature parameters. We could add them all by hand, but that would be really tedious. Facebooker provides a helper to make this easier. If we include the `Facebooker::Rails::TestHelpers` module in our tests, we'll get some additional helper methods. One of these methods, `facebook_get`, will fill in the `fb_sig` parameters for us.

When testing actions that are expected to be called inside the Facebook canvas, you should always use the `facebook_*` version of the request method. So, `get` becomes `facebook_get`, and `post` becomes `facebook_post`:

```
include Facebooker::Rails::TestHelpers
def test_new_invitation
  facebook_get :new
  assert_response :success
  assert_template 'new'
end
```

Our test passes, but it isn't clear exactly what is happening. We know that `facebook_get` generates a signature for our request, but what parameters are included? By default, all the Facebooker helpers simulate a page request for the user with Facebook ID 1234 viewing a canvas page. We can override those defaults by sending in our own parameters. For instance, to make sure our new action requires a user to be logged in, we could write a test that passes in `nil` for the user. Let's write a test that makes sure that happens:

```
def test_get_new_requires_user
  facebook_get :new, :fb_sig_user=>nil
  assert_response :redirect
end
```

Running that, we get the error "Expected response to be a `<fb:redirect>`, but was `<200>`" That's caused by the way Facebook handles redirects. Instead of sending a special HTTP status code, Facebook redirects use the `<fb:redirect>` tag.

We haven't talked about `<fb:redirect>` yet because Facebooker handles it for us. We will need to think about it during our tests. Instead of testing for a redirect with the normal `assert_response :redirect` code, we instead want to use `assert_facebook_redirect_to`. We will make sure our user is redirected to the authorization path for our application that can be obtained by calling `login_url` on a `Facebooker::Session` object. That makes our finished version of the test look like this:

```
def test_get_new_requires_user
  facebook_get :new, :fb_sig_user=>nil
  assert_facebook_redirect_to Facebooker::Session.create.login_url
end
```

We can do more than just test `get` requests inside Facebook. Facebooker also provides `facebook_post`, `facebook_put`, and `facebook_delete` methods for use inside our tests.

Let's use the `facebook_post` method to make sure our invitation's `create` method works:

```
def test_valid_create
  facebook_post :create, :ids=>["1234"]
  assert_response :success
  assert_template 'create'
end
```

Excellent. That test passes and feels much more like a typical Rails controller test. There's just one more thing we need to test. Let's make sure a user's profile is updated when they click the link in our invite. That's going to cause a problem. We don't want to actually update a user's profile, but we want to make sure an attempt is made. We'll need a new technique to test that.

Using Mocks and Stubs

We've successfully modified our tests to generate the correct Facebook signature to let us test our controllers without Facebook. We got stuck trying to verify that a profile update was happening. We need to verify that our profile is being updated, but we don't want our tests to actually talk to Facebook. Not only will that make them run slowly; it also makes them less reliable. We could do something ugly, such as wrap the code that talks to Facebook inside an `if RAILS_ENV!="test"` statement. That doesn't give us much confidence that our code does what we want. Instead, we can use *mocks* and *stubs*.

If you haven't heard of mocks and stubs, you aren't alone. Mock objects are a relative newcomer onto the testing scene. In fact, the first paper that described them wasn't even written until the year 2000.¹ Mocks and stubs are objects that can be used to replace functionality after the fact. For instance, in our new action test, we could use a stub object to replace the Facebook user. Our user stub accepts the same calls as the real user but eliminates the side effects. This will allow us to test the `hometown` method without needing to talk to Facebook.²

Several Ruby libraries make it easy to use mock and stub objects in our code. We're going to use FlexMock from Jim Weirich in our examples. Before we go any further, you'll need to install the FlexMock gem

1. Mocks were first described in <http://www.mockobjects.com/files/endotesting.pdf>.

2. You can read more about testing with mocks and stubs at <http://www.ibm.com/developerworks/web/library/wa-mockrails/index.html>. That article uses a different mock library than we will, but the concept is the same.

by running `gem install flexmock`. After installing the gem, we'll need to include FlexMock into our tests by adding a `require` to our code, as shown here:

```
require File.dirname(__FILE__) + '/../test_helper'  
require 'flexmock/test_unit'  
class InvitationsControllerTest < ActionController::TestCase  
  ...  
end
```

The idea of mocks and stubs is probably still a little vague. Let's make it concrete by looking at an example. Let's write a test for our new action where we specify the `from` parameter:

```
def test_new_with_from_updates_profile  
  facebook_get :new, :from=>1  
  assert_response :success  
end
```

When that code runs, it makes a call to Facebook to update a profile, just the thing we want to avoid. We need some way to stop this from happening. We'll use FlexMock to replace the call to `profile_fbml=` with a method that just returns `true`. To do this, we just add a single line to the beginning of the test:

```
def test_new_with_from_updates_profile  
  flexmock(@controller, :update_profile=>true)  
  facebook_get :new, :from=>1  
  assert_response :success  
end
```

The call to `flexmock` creates a *stub*. A stub is used to replace the side effects from calling a method. In this particular example, calling `@controller.update_profile` will no longer update a user's profile. Instead, it will just return `true`. Stubs replace functionality only inside the test in which they are used. Running that gets our tests to pass, but we want to do more than just remove calls to the Facebook API. We also want to verify that the calls are made correctly. To do that, we will use mocks.

Mock objects replace a method's implementation, but they also verify that the call was made. This verification is the main difference between a mock object and a stub.³ To change our stubs into mocks, we just need to add an expectation. An expectation tells the mock object what methods should be called and can optionally give constraints about the parameters sent to each method. In FlexMock, we use the `should_receive`

3. Martin Fowler wrote a great article about the differences between mocks and stubs. You can find it at <http://www.martinfowler.com/articles/mocksArentStubs.html>.

method to add constraints. For instance, we could make sure that our `update_profile` method is called with the following code:

```
def test_new_with_from_updates_profile
  flexmock(@controller).should_receive(:update_profile)
  facebook_get :new, :from=>1
  assert_response :success
end
```

If our code doesn't call `update_profile`, FlexMock will raise an exception, as you can see here:

```
test_new_with_from_updates_profile(InvitationsControllerTest)
 [flexmock-0.8.0/lib/flexmock/validators.rb:40:in `validate'
  flexmock-0.8.0/lib/flexmock/expectation.rb:123:in `flexmock_verify'
  flexmock-0.8.0/lib/flexmock/expectation.rb:122:in `each'
  flexmock-0.8.0/lib/flexmock/expectation.rb:122:in `flexmock_verify'
  flexmock-0.8.0/lib/flexmock/expectation_director.rb:61:in `flexmock_verify'
  flexmock-0.8.0/lib/flexmock/expectation_director.rb:60:in `each'
  flexmock-0.8.0/lib/flexmock/expectation_director.rb:60:in `flexmock_verify'
  flexmock-0.8.0/lib/flexmock/core.rb:76:in `flexmock_verify'
  flexmock-0.8.0/lib/flexmock/core.rb:75:in `each'
  flexmock-0.8.0/lib/flexmock/core.rb:75:in `flexmock_verify'
  flexmock-0.8.0/lib/flexmock/core.rb:191:in `flexmock_wrap'
  flexmock-0.8.0/lib/flexmock/core.rb:74:in `flexmock_verify'
  flexmock-0.8.0/lib/flexmock/mock_container.rb:40:in `flexmock_verify'
  flexmock-0.8.0/lib/flexmock/mock_container.rb:39:in `each'
  flexmock-0.8.0/lib/flexmock/mock_container.rb:39:in `flexmock_verify'
  flexmock-0.8.0/lib/flexmock/mock_container.rb:31:in `flexmock_tear_down'
  flexmock-0.8.0/lib/flexmock/test_unit.rb:26:in `tear_down_without_fixtures'
  activerecord-2.0.2/lib/active_record/fixtures.rb:987:in `full_tear_down'
  activesupport-2.0.2/lib/active_support/testing/default.rb:7:in `run']]:
in mock 'flexmock(InvitationsController)': method 'update_profile(*args)' ...
<1> expected but was
<0>.
```

That verifies that the `update_profile` method in our controller was called, but it doesn't do anything to verify the content that was sent. Let's change our test to verify that the `profile_fbml=` method was called. To do this, we'll need to somehow get access to the `Facebooker::User` object that is created by the controller. We can do this by using the `new_instances` method of FlexMock. The `new_instances` method allows us to mock or stub calls on all instances of a class, something that is very convenient for objects that are created during the request:

```
def test_new_with_from_updates_profile
  flexmock(Facebooker::User).new_instances.
    should_receive(:profile_fbml=).once
  facebook_get :new, :from=>1
  assert_response :success
end
```


That lets us verify that `profile_fbml=` is called but doesn't help us check the value passed. To do that, we'll use a *matcher*. Matchers in mock object terms are constraints that are verified on calls to mock objects. We've already used one matcher when we called `once` in our previous test. To verify the parameters passed to a method, we can add the `with` matcher. When we call `with`, we give it the parameters that we expect our method to receive. In this case, that is the profile content:

```
def test_new_with_from_updates_profile
  profile_expectation =
    "<fb:fbml>\n I was sent here by <fb:name uid='1' />\n</fb:fbml>"
  flexmock(Facebooker::User).new_instances.
    should_receive('profile_fbml=').\
    once.with(profile_expectation)
  facebook_get :new, :from=>1
  assert_response :success
end
```

Give that a try. If our code either doesn't call `profile_fbml=`, calls it more than once, or calls it with the wrong content, FlexMock will raise an exception, and our test will fail.

Now we've verified that our controller code works. We've even checked to make sure our interaction with Facebook happens as we would expect. Next, we'll turn our attention to testing our models.

4.2 Testing Models

Our controller tests looked quite different from a typical Rails test. Luckily, our model tests should feel much more familiar. For the most part, our model tests will work just like they normally would. There will be a few areas where we'll need to use mocks and stubs to isolate our tests from Facebook.

When we look at our models, there are only a small number of areas that will need to be isolated. We saw an example of stubbing a call to Facebook earlier. Let's use that same idea to test our `attack` creation and `hometown` method.

We will start our model tests the same way we would for any Rails application—by making sure we have the right validations in place on our `Attack` object. We can complete this with four quick tests. First, we'll create a test to make sure an attack is valid when all required fields are present. Next, we'll test each validation by omitting a required field and making sure the object is not valid.

```

fixtures :users, :moves
def setup
  @attack = Attack.new(:attacking_user=>users(:jen),
                      :defending_user=>users(:mike),
                      :move=>moves(:chop))
end

def test_valid
  assert @attack.valid?
end

def test_attack_requires_attacking_user
  @attack.attacking_user=nil
  assert !@attack.valid?
end

def test_attack_requires_defending_user
  @attack.defending_user=nil
  assert !@attack.valid?
end

def test_attack_requires_move
  @attack.move=nil
  assert !@attack.valid?
end

```

When we run these, we get some errors. It looks like we didn't add validations earlier. Let's add those now:

```

class Attack < ActiveRecord::Base
  validates_presence_of :attacking_user_id, :defending_user_id, :move
  ...

```

With that done, let's create some tests for our hometown method. We're going to need to use a stub to bypass the call to Facebook. This will require us to include FlexMock in that test. Instead of adding an include line to every test file, we can add our include in `test/test_helper.rb`. We can also add our Facebooker test helper include there, as shown here:

```

ENV["RAILS_ENV"] = "test"
require File.expand_path(File.dirname(__FILE__) +
  "../config/environment")
require 'test_help'
require 'flexmock/test_unit'

class Test::Unit::TestCase
  include Facebooker::Rails::TestHelpers
  ...

```

Now that we've done that, let's get on with our testing. Testing our hometown method will be a little different from what we've done before. Not only do we need to remove the call to Facebook, but we also need FlexMock to give us a specific return value. We can do this with the `and_return` method. To use `and_return`, we'll first need to create the object we want returned. Then, we'll stub the `hometown_location` method on the user and tell FlexMock to return our newly created value. It sounds more complicated than it is:

```
def test_hometown_when_exists
  location=Facebooker::Location.new(:city=>"Westerville",
                                   :state=>"Ohio")

  fm=flexmock(Facebooker::User)
  fm.new_instances.should_receive(:hometown_location).
    and_return(location)
  mike=users(:mike)
  assert_equal "Westerville Ohio",mike.hometown
end
```

Now that we have complete control over the data returned by the Facebook API call, we can easily test other situations. For example, we could make a small change to test what happens when no hometown is provided by simply changing the return value:

```
def test_hometown_when_blank
  location=Facebooker::Location.new(:city=>"", :state=> "")
  fm=flexmock(Facebooker::User)
  fm.new_instances.should_receive(:hometown_location).
    and_return(location)
  mike=users(:mike)
  assert_equal "an undisclosed location",mike.hometown
end
```

You'll need to be careful when testing with mocks. If the actual API call returns something different from what your mock does, you may be building a false sense of security. For example, in our previous example, Facebook returns a location with a blank city and state to signify an unknown location. If we instead made our mock return a `nil` location, we could make our tests pass even though our application would be broken when used with Facebook.

Using the techniques we've seen so far, you should be able to write the rest of your model tests, testing things like belt upgrades and available moves.

4.3 Summary

We've now looked at how to test all the parts of our Facebook application. We've seen how to test Facebook controller requests using both Facebooker helpers and mocks. We've looked at testing our models as well. Obviously we just scratched the surface in terms of actually writing tests, but we did cover all the concepts. Even though we won't talk about it in future chapters, all the code samples included with the book will include tests. You can look at these tests along with the code to see more examples of testing a Facebook application.

Now that we have confidence our application is functioning as we intend it to, let's turn our attention to building the interface for Karate Poke.

Chapter 5

Getting Into the Facebook Canvas

We've built a data model for our application and seen how we can write tests for it. Now it's time to start building the web interface for Karate Poke. We'll start by looking at forms in Facebook. From there, we'll see some tools for adding navigation. We'll finish by adding pagination and giving Karate Poke some style. Along the way we'll use many of the FBML primitives. This isn't meant to be an exhaustive FBML reference. You can find that on the developer wiki.¹

5.1 Getting Interactive with Forms

We're going to start our tour of the Facebook canvas with a topic that is central to most web applications: the web form. I know it isn't the sexiest topic, but it's one of our only tools for getting information from our users.

Let's make a form for initiating attacks. Before we can do that, let's create an attacks controller. We'll be spending most of our time working on the `new()`, `create()`, and `index()` methods. You can create the controller and the methods all at once by running this:

```
$ script/generate controller attacks new create index
```

With our controller created, we can turn it into a RESTful resource using `map.resources`:

```
Download chapter5/karate_poke/config/routes.rb
```

```
map.resources :attacks
```

Now we're ready to create our form.

1. <http://wiki.developers.facebook.com/index.php/FBML>

Building Our First Form

We created our `User#attack` method in Section 3.4, *Attack!*, on page 60. To call it, we'll need attacking and defending users as well as a `Move` object. Let's look at how we can get that information.

We can get the attacking user from the `current_user()` helper that we built previously. That means our form needs to provide only the defending user and the move. We could create something simple like this:

```
<% form_for Attack.new do |f| %>
  Move: <%= f.collection_select :move_id,
    current_user.available_moves, :id, :name %> <br />
  User to attack: <%= f.text_field :defending_user_id %> <br />
  <%= submit_tag 'Attack!' %>
<% end %>
```

You can put that in `app/views/attacks/new.fbml.erb`. Notice that the file-name ends in `.fbml.erb`. Rails looks for different files depending on the content type requested. For HTML, it will look for `.html.erb`. For JavaScript, it will look for `.js.erb`. Because Facebook views are written in FBML, Rails is looking for an `.fbml.erb` file. Earlier, we used just `.erb`. If Rails can't find a file for the requested content type, it will look for a plain `.erb` file.

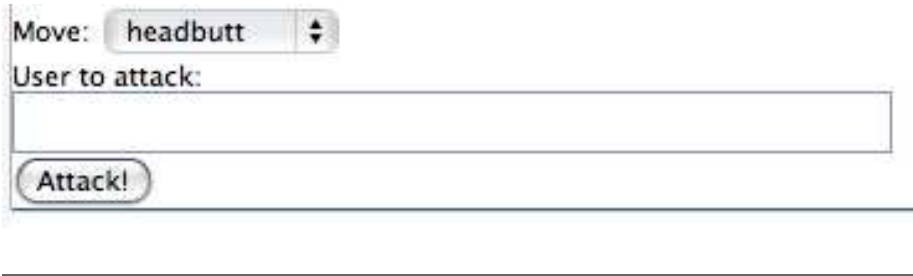
Although that works, it is hard to use. We saw a much nicer way of selecting Facebook users in Section 2.2, *Creating the Invitation Form Using FBML*, on page 40. Unfortunately, Facebook allows the graphical friend picker to be used only in request forms. Facebook does provide a text-based friend picker that we can use in any form:

```
<% form_for Attack.new do |f| %>
  Move: <%= f.collection_select :move_id,
    current_user.available_moves, :id, :name %> <br />
  User to attack: <fb:multi-friend-input /> <br />
  <%= submit_tag 'Attack!' %>
<% end %>
```

You can see our attack form in Figure 5.1, on the next page. Make sure you test the typeahead features of the `<fb:multi-friend-input>` tag. Like the `<fb:request-form>` tag we saw in Section 2.3, *Giving the Sender Some Feedback*, on page 43, the `<fb:multi-friend-input>` will cause Facebook to send the user IDs of selected users in the `ids` parameter.

Implementing the Attack

With our attack form in place, we need to build a controller action that calls the `attack()` method. We can use the `User.for` method we built to turn each provided Facebook ID into a `User` object.



The image shows a web form with a label 'Move:' followed by a dropdown menu containing the text 'headbutt'. Below this is a text input field with the label 'User to attack:'. At the bottom of the form is a button labeled 'Attack!'.

Figure 5.1: Our first form

Let's use that to add a create action to our attacks controller:

```
Line 1 def create
-   attack = Attack.new(params[:attack])
-   for id in params[:ids]
-     current_user.attack(User.for(id), attack.move)
5   end
-   redirect_to new_attack_path
- end
```

Let's walk through that code. First, on line 2, we create a new Attack object from our form parameters. This gives us the Move that was selected from the form. On lines 3 through 5, we loop through the Facebook ids parameters that Facebook sends us. We turn each one into a User object using our for() method. On line 4, we call the attack() method we built.

When I use our form, I can see that the attacks are created in the database, but I don't get any feedback. Let's add a message to tell how many hits and misses there were.

First, we'll need to separate the hits from the misses. We can just create an array for each. (You could also use Enumerable#partition for this.)

```
def create
  attack = Attack.new(params[:attack])
  hits = []
  misses = []
  for id in params[:ids]
    attack = current_user.attack(User.for(id), attack.move)
    if attack.hit?
      hits << attack
    else
      misses << attack
    end
  end
  redirect_to new_attack_path
end
```

Your attack resulted in 0 hits and 1 miss.

Figure 5.2: Facebook success messages are displayed in yellow.

Now we can build our message. We use `(hits.size==1 ? "hit" : "hits")` instead of the more idiomatic `pluralize` because our code is in the controller. The Rails `pluralize` helper is available only in views.

```
...
flash[:notice] = "Your attack resulted in #{hits.size} " +
  (hits.size==1 ? "hit" : "hits") +
  " and #{misses.size} " +
  (misses.size == 1 ? "miss" : "misses") + "."
redirect_to new_attack_path
```

With our message sorted out, we need a way to display it. Facebook has a standard way of displaying messages, as you can see in Figure 5.2. We can use the `<fb:success>` FBML tag to make our messages look like this. Let's add that to our view:

```
<% unless flash[:notice].blank? %>
<fb:success>
  <fb:message>
    <%= flash[:notice] %>
  </fb:message>
</fb:success>
<% end %>
<% form_for :attack, @attack, :url=>create_attack_path do |f| %>
```

That's a lot of work to display a message. Facebooker provides the `facebook_messages()` helper to display the flash as a Facebook message:

```
<%= facebook_messages %>
<% form_for :attack, @attack, :url=>create_attack_path do |f| %>
...

```

Along with showing notice messages, the `facebook_messages()` helper will also show error messages that are stored in `flash[:error]`. You can see an example error message in Figure 5.3, on the following page. Let's add an error message if our user doesn't select any users to attack:

```
def create
  if params[:ids].blank?
    flash[:error] = "You forgot to tell me who you wanted to attack!"
  else
    attack = @attack.new(params[:attack])
    hits = []
  end
end
```


You forgot to tell me who you wanted to attack!

Figure 5.3: Facebook error messages are displayed in red.

```

    misses = []
    ...
  end
  redirect_to new_attack_path
end
  ...

```

We have our form in place, and it's working well, but it doesn't really match the Facebook look and feel. Let's fix that next.

Matching the Facebook Interface

Since we're building an application that sits in the middle of Facebook, we should really make it blend in. We've already used the `<fb:message>` and `<fb:multi-friend-input>` tags to make our application look like Facebook. Let's use another FBML tag to make our form look more like a Facebook form.

To do that, we'll need to use the `<fb:editor>` FBML tag. The `<fb:editor>` tag is a replacement for the HTML `<form>` tag. It takes many of the same parameters as the `<form>` tag such as action and method. Inside, you can add form fields using other Facebook-specific tags, such as `<fb:editor-text>` for text fields.

Here's what the code looks like using the `<fb:editor>` tag. You can see the result in Figure 5.4, on the next page.

```

<fb:editor action="<%=attacks_path%>">
  <fb:editor-custom label="Move:">
    <%= select_tag "attack[:move_id]",
      options_from_collection_for_select(
        current_user.available_moves,:id,:name) %>
  </fb:editor-custom>
  <fb:editor-custom label="User to attack:">
    <fb:multi-friend-input />
  </fb:editor-custom>
  <fb:editor-buttonset>
    <fb:editor-button value="Attack!"/>
  </fb:editor-buttonset>
</fb:editor>

```



Figure 5.4: The `<fb:editor>` form matches the Facebook style.

It looks great in the browser, but that code feels awkward. Rails provides us with nice tools such as form builders to work with HTML forms. I hate to lose those helpers just to match the Facebook interface. Facebooker provides a helper to make it easy to create `<fb:editor>` forms. Let's change our attack form:

```
<% facebook_form_for :attack, Attack.new, :url=>attacks_path do |f| %>
  <%= f.collection_select :move_id,
    current_user.available_moves, :id,
      :name, :label=>"Move" %> <br />
  <%= f.multi_friend_input :label=>"User to attack"%>
  <%= f.buttons 'Attack!' %>
<% end %>
```

This version looks the same in the browser, but our code is much cleaner. We replaced all the custom FBML with a call to `facebook_form_for()`. We replaced our `<fb:multi-friend-input>` tag with a call to the form builder's `multi_friend_input` method. By using the Facebook form builder, we built a Facebook-specific form without losing the beauty of Rails.

You can use `facebook_form_for` just like you would use `form_for`. Along with all the Rails form helpers, Facebooker adds some helpers specific to Facebook. You can find the documentation for these helpers on the Facebooker website.²

Now that we have a working attack form that looks like part of Facebook, let's move on.

2. <http://facebooker.rubyforge.org>

5.2 Building the Battles Page

With our attack form done, we need to build a landing page. Let's create a page that shows our users' recent battle history.

Our battles page shows a list of attacks, so let's use the index action on our attacks controller. We want this to be our application's main page, so let's make it the default route:

```
Download chapter5/karate_poke/config/routes.rb
```

```
map.battles '', :controller=>"attacks", :action=>"index"
```

We built most of the logic for our battles page when we built our User model:

```
def index
  @battles = current_user.battles
end
```

With our action in place, we can create a view. For each battle, we want to show the image, the name of the attacker and defender, and whether the attack was a hit or miss:

```
<% for attack in @battles %>
<div class="battle">
  <%= image_tag attack.move.image_name %>
  <%= fb_name attack.attacking_user %>
  <%= attack.hit? ? "hit" : "missed" %>
  <%= fb_name attack.defending_user %>
  with a <%= attack.move.name %>
</div>
<% end %>
```

You can see what that looks like in Figure 5.5, on the following page.

If you've attacked a few people, our battles page looks good. If you haven't, you will see only a blank page. That's not very inviting. Let's instead direct new users to our attack page. We can include a flash message to encourage them to attack somebody:

```
def index
  @battles = current_user.battles
  if @battles.blank?
    flash[:notice]="You haven't battled anyone yet."+
      " Why don't you attack your friends?"
    redirect_to new_attack_path
  end
end
```



Figure 5.5: Our battles page

That looks much better for new users. Now, let's focus on users who already have some attacks. You'll see that each name has a link we didn't add. By default, the `<fb:name>` tag creates a link from each user's name to their profile page. We don't want to link to a user's profile page; we want our links to go to that user's battles page.

Facebook doesn't give us a way to change the destination of the link. It does, however, let us remove it entirely. We do this by adding the `:linked => false` option on our call to `fb_name()`. Once we've removed the profile link, we can add one of our own:

```
<%= link_to(
  fb_name(attack.attacking_user, :linked=>false),
  battles_path(:user_id=>attack.attacking_user)) %>
<%= attack.hit? ? "hit" : "missed" %>
<%= link_to(
  fb_name(attack.defending_user, :linked=>false),
  battles_path(:user_id=>attack.defending_user)) %>
```

Now we have our links pointing to the battles page, but our index action shows only the battles of the currently logged in user. We need to modify our `index()` action to use the `user_id` parameter to display another user's battles:

```
def index
  if params[:user_id]
    @user = User.find(params[:user_id])
  else
    @user = current_user
  end
  @battles = @user.battles
  ...
end
```

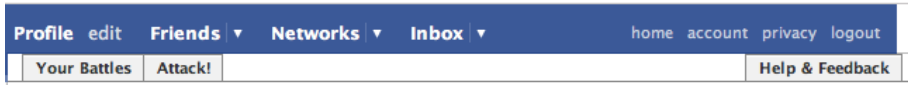


Figure 5.6: It's easy to create a tab bar with `<fb:tabs>`.

With our links squared away, let's add a little more information to the battles page. Right now, you can't tell whose page you are looking at. Let's add the owner's name and their profile picture:

```
<h2> <%= fb_name @user, :useyou=>false,
:linked=>false, :possessive=>>true%>
Battle History </h2>
<%= fb_profile_pic @user %>
```

We used a lot of options on our call to `fb_name()`. You can see all the available options on the developer wiki.³

We have an attack form and a battle page, but now we have a new problem. We have two different pages and no navigation.

5.3 Adding Navigation

Many Facebook applications use a tab bar for navigation. Let's stay with the Facebook look and feel and use one as well. If you're the kind of person who has nightmares about the CSS required to build a tab navigation bar, don't despair. Facebook has done the work for us.

Adding a Tab Bar

Earlier, we used the `<fb:editor>` tags to make our form match the Facebook look and feel. Here, we can use two new FBML tags, `<fb:tabs>` and `<fb:tab-item>`, to add a tab bar to our application. You can see the end result in Figure 5.6.

To create a tab bar, you just place one or more `<fb:tab-item>` tags inside an `<fb:tabs>` tag. The tab item takes two parameters, one for the text of the tab and the other for the link. Let's use the raw FBML and give it a try.

3. <http://wiki.developers.facebook.com/index.php/Fb:name>

Add this to the top of your new attack view:

```
<fb:tabs>
  <fb:tab-item title="Your Battles" href="<%=battles_path%" />
  <fb:tab-item title="Attack!" href="<%=new_attack_path%" />
</fb:tabs>
<% facebook_form_for Attack.new do |f| %>
...

```

I feel a little dirty when I write FBML by hand. Let's use the Facebooker helpers to clean that up:

```
<% fb_tabs do %>
  <%= fb_tab_item "Your Battles", battles_path %>
  <%= fb_tab_item "Attack!", new_attack_path %>
<% end %>
<% facebook_form_for Attack.new do |f| %>
...

```

That looks good, but you can't tell which tab is active. Let's add a parameter to tell Facebook to highlight the active tab.

```
<% fb_tab do %>
  <%= fb_tab_item "Your Battles", battles_path %>
  <%= fb_tab_item "Attack!", new_attack_path, :selected => true %>
<% end %>
<% facebook_form_for Attack.new do |f| %>
...

```

Let's add tabs to our battles page as well:

```
<% fb_tab do %>
  <%= fb_tab_item "Your Battles", battles_path, :selected => true %>
  <%= fb_tab_item "Attack!", new_attack_path %>
<% end %>
<% for attack in @battles %>
...

```

I'm noticing a lot of duplication in our pages. That isn't very DRY. By DRY, I mean Don't Repeat Yourself, a phrase coined in the excellent *The Pragmatic Programmer: From Journeyman to Master* [HT00]. This is the idea that every concept should be expressed just once. To clean it up, let's move our tabs into the application layout. When we do that, we'll need a way to know which tab is active.

Right now, we have only two different tabs. We know the battles tab has a @battles attribute, so let's use it to distinguish between the two cases. If the @battles attribute is not nil, we know we're on the battles page. If @battles is nil, we know we're on the attack tab.

Let's create an `app/views/layouts/application.fbml.erb` file and add the following code to it. We created an `application.erb` layout earlier. We're using the more specific `application.fbml.erb` now so that our layout is used only for requests from Facebook. This will come in handy in Chapter 8, *Integrating Your App with Other Websites*, on page 158. We're also changing our `<fb:fbml>` tag to include the version attribute. This tells Facebook that we want to use all the newest features of FBML.

```
<fb:fbml version="1.1">
<% fb_tabs do%>
  <%= fb_tab_item "Your Battles", battles_path,
                :selected=>!@battles.nil? %>
  <%= fb_tab_item "Attack!", new_attack_path,
                :selected=>@battles.nil? %>

<% end %>
<%= yield%>
</fb:fbml>
```

That eliminates the duplication nicely. Now we'll have to make a change in only one place to add a new tab.

Linking to the About Page

Facebook provides an about page for every application to which the application directory links. You can use your about page to describe your application to potential users. It also has an area for reviews and a discussion board for your users. Let's add a link to the help page on the right side of our tab bar:

```
<% fb_tabs do%>
  <%= fb_tab_item "Your Battles", battles_path,
                :selected=>!@battles.nil? %>
  <%= fb_tab_item "Attack!", new_attack_path,
                :selected=>@battles.nil? %>
  <%= fb_tab_item "Help & Feedback",fb_about_url,:align=>"right"%>
<% end %>
```

There are a couple of new things in this link. Facebooker provides the `fb_about_url()` method to get a link to our application's about page. We also used the `align` option to the `fb_tab_item()` method to move our tab to the right.

I wish creating a tab navigation bar was always this easy. In just five lines of code we added a complete navigation system to our application.

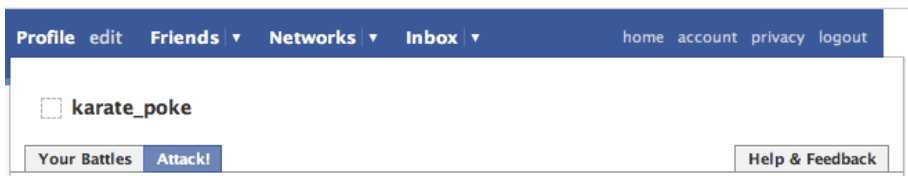


Figure 5.7: The Facebook dashboard displays the application's name and image.

Adding a Header

The tab navigation is an improvement, but it still doesn't look right. Our tab bar is too close to the top of the page. It'd be nice to add a header to the page. Facebook provides the `<fb:dashboard>` tag to display application header information. Let's add a dashboard to our layout.

We'll skip right to the Facebooker helpers here. You can see more information about the `<fb:dashboard>` tag on the developer wiki.⁴

```
<fb:fbml>
  <%= fb_dashboard %>
  <% fb_tabs do%>
    ...
```

That's all it takes to add a dashboard. You can see the end result in Figure 5.7.

The empty square is a placeholder for the logo of our application. You can upload an icon for your application using the Facebook Developer application. You can see that Facebook includes our application name in the dashboard automatically.

Along with our application name and image, we can also add actions to our dashboard. Let's add a link to the invitation page we created earlier. We'll need to change our `fb_dashboard()` call to use a block and add a call to the `fb_action()` method inside:

```
<% fb_dashboard do %>
  <%=fb_action "Invite your friends",new_invitation_path%>
  <%=fb_help "Help & Feedback", fb_about_url%>
<% end %>
```

4. <http://wiki.developers.facebook.com/index.php/Fb:dashboard>



Figure 5.8: A dashboard with an action and a help link

Along with actions, dashboards can also contain a help link. By default, help links are placed on the right side of the dashboard. You can see what this looks like in Figure 5.8. The `<fb:dashboard>` acts as an alternative to the `<fb:tab>` area for navigation. I prefer the tabs for navigation, so let's remove the help link and action from our dashboard.

We've made some really small changes to our application that add a lot of style. We'll continue to look at other ways of adding style to our application throughout the rest of this book. Let's move on to looking at some more functional parts of the Facebook canvas.

5.4 Hiding Content from Users

As we develop Karate Poke, we'll run into situations where we want to show content to only certain users. Facebook provides a number of powerful tools to make it easier to handle these cases. We'll see an example of how to use these tools here.

We're going to make it easy to attack a user from their battles page. The Facebook user selector allows you to enter only those people you are friends with, so this will also give us a way to attack people who aren't our friends. We'll add a form to the top of the page that allows the viewer to attack the owner of a battles page. We don't want this form to show up when we view our own battles page. After all, we wouldn't want to attack ourselves.

We already have the create action working, so let's build a form that will reuse it. We can add a hidden field that includes the user's Facebook ID. If we name the field `ids[]`, we won't have to change our create action. (By adding `[]` to the end of a form field name, you tell Rails that you want the result to be an array.)

Become Part of Facebook

When you build a Facebook Platform application, you have powerful tools at your disposal. Facebook has gone beyond just providing an API for accessing its data. It also gives you access to the same tools it uses in its applications. In fact, the integration is so deep that you can compete with Facebook's own applications!

Take FunWall, for example—the most popular externally developed Facebook application. On a typical day, more than 5 million people use FunWall. That's amazing, since FunWall competes directly with an application that comes already installed. Created by Slide, FunWall has done more than just duplicate all the features of part of Facebook. It has also added enough new features to be installed by one out of every three Facebook users. The creators of FunWall have taken full advantage of the Facebook platform to provide an immersive user experience. Many of FunWall's users don't even know that the application wasn't built by Facebook.

Your code is just as much a part of Facebook as the applications developed by Facebook. You have access to the same actions, interface, and messaging that Facebook uses. You can send emails to your users and write to their profiles.

FBML is a powerful tool that allows you to match the Facebook user interface. By keeping your application consistent with the Facebook UI, you reduce the learning curve for your application. When a new user sees your app, they will immediately know how things work, much like a Macintosh user using a new application. Maintaining a consistent UI is a powerful way to reduce the cost of using your application.



Figure 5.9: Looking at your own battles page

```
<h2> <%= fb_name @user, :useyou=>false,
:linked=>false, :possessive=>true%> Battle History </h2>
<%= fb_profile_pic @user %>
<hr />
<h3>Do you want to attack <%= fb_name @user%>?</h3>
<% facebook_form_for Attack.new do |f| %>
  <%= f.collection_select :move_id,
  current_user.available_moves, :id, :name, :label=>"Move" %> <br />
  <%= hidden_field_tag "ids[]", @user.facebook_id%>
  <%= f.buttons "Attack!" %>
<% end %>
...

```

With that working, we just need to hide the form when a user views their own battles page (Figure 5.9).

In a normal Rails application, we would probably fix this by adding an unless statement around the form, something like this:

```
<% unless @user == current_user %>
...
<% end %>

```

That would work, but Facebook gives us another possibility. Let's use the FBML `<fb:if-is-user>` tag instead. Let's look at how the two statements work.

The `<fb:if-is-user>` tag acts like a normal if statement that is executed when Facebook renders the page. That means your application will output the same data no matter who the viewer is. Facebook will decide whether to show the form when it processes the page. The unless statement, on the other hand, sends different data to Facebook depending upon who the viewer is.

The result is the same for our viewer, but by using the `<fb:if-is-user>` tag, we can allow our application to cache the page. We'll talk more about caching in Section 9.2, *Caching Our Views*, on page 173.

Here's what the FBML version looks like:

```
<% fb_if_is_user @user do %>
  <% fb_else do %>
    <hr />
    <h3>Do you want to attack <%= fb_name @user%>?</h3>
    <% facebook_form_for Attack.new do |f| %>
      <%= f.collection_select :move_id,
        current_user.available_moves, :id, :name, :label=>"Move" %>
      <br />
      <%= hidden_field_tag "ids[]", @user.facebook_id%>
      <%= f.buttons "Attack!" %>
    <% end %>
  <% end %>
<% end %>
```

Because FBML is an XML variant, there is no way to do the traditional if, else, end sequence. You have to nest the else statement inside the if block. This looks awkward at first, but you will quickly get used to it.

Along with `<fb:if-is-user>`, Facebook provides a number of other if tags. For example, we can use the `<fb:if-is-friends-with-viewer>` tag to show content only if a user is friends with the viewer. We could use the `<fb:is-in-network>` tag to limit content to a specific network. We won't use these in Karate Poke, but it's nice to know they exist.

Take a look at your battles page now. Sometimes when I view a user's battles page, it looks a little strange. It looks like the person's name is missing, as you can see in Figure 5.10, on the next page.

We've talked about the Facebook privacy implementation in the past. Here's the first time we're seeing it come into play. In Figure 5.10, on the following page, I'm logged in as myself while trying to view the battles page of a test user. Since test users aren't visible to regular users, the `<fb:name>` tag is displaying a blank name. This can also happen between two regular users if one of them has very strict privacy settings. That looks bad—let's fix it before we continue.

We've specified several different options to the `<fb:name>` FBML tag. We'll use yet another option to tell Facebook what to display when a username would normally be hidden.



Figure 5.10: Our battles page between nonfriends

Let's use the phrase "a hidden ninja" in place of the empty string:

```
...
<h3>Do you want to attack
  <%= fb_name @user, :ifcantsee=>"a hidden ninja"%>?</h3>
...
```

We'll need to add that parameter to all the other places we use the `<fb:name>` tag. Adding that option to all our calls to `fb_name()` isn't very DRY. We should create a helper to do this for us. Let's add it to `app/helpers/application_helper.rb`. While we're at it, let's also add a helper to tell us whether an attack was a hit or a miss.

```
def name(user,options={})
  fb_name(user,{:ifcantsee=>"a hidden ninja"}.merge(options))
end

def attack_result(attack)
  attack.hit? ? "hit" : "missed"
end
```

That looks much better. With all the testing we've been doing, our battles page is getting really long. Our battles page would look a lot more polished if it had multiple pages instead of just a long list of battles.

5.5 Adding Pagination

Pagination in a Facebook application isn't any different than in other Rails applications. Because of the large numbers of users you may have, the performance of our pagination implementation is incredibly

important. The old Rails pagination helpers performed poorly⁵ and were removed in Rails 2.0.

In place of the old pagination helpers, let's use the *will_paginate* plugin.⁶ To enable pagination in our models, we're going to need to make a small change. In our `User#battles` method, we need to replace our call to `find()` with a call to `paginate()`. We'll also need to add a parameter to accept the page number.

Download `chapter5/karate_poke/app/models/user.rb`

```
def battles(page=1)
  page ||= 1
  Attack.paginate(
    :conditions=>
      ["attacking_user_id=? or defending_user_id=?",
       self.id,self.id],
    :include=>[:attacking_user,:defending_user,:move],
    :order=>"attacks.created_at desc",
    :page => page,
    :per_page => 5)
end
```

We're using five attacks per page so that we get multiple pages more quickly during development. We may want to change that to a larger number before we launch.

OK, now that we have that setup done, give it a try in `script/console`. Now you can page through your battles. We just need to update our controllers and view. Let's modify our controller to pass the page number parameter to our `battles` method:

```
def index
  ...
  @battles = @user.battles(params[:page])
  ...
end
```

That's all the setup we need to do with our controllers. With that done, we can update our views. We'll use the `will_paginate` helper method to render paging links for our battles:

```
...
<%= will_paginate(@battles) %>
<% for attack in @battles %>
  <div class="battle">
  ...
```

5. <http://glu.ttono.us/articles/2006/08/30/guide-things-you-shouldnt-be-doing-in-rails>

6. Install it by running `script/plugin install svn://errtheblog.com/svn/plugins/will_paginate`.

That's all it takes to get our battle list paginating. I love how easy `will_paginate` makes things. I'm not so sure I like the pagination links on the left side of the screen, however. To get them looking better, we'll need to use CSS. We'll look at that next.

5.6 Adding Some Style

Karate Poke is starting to come together. It's still lacking in the style department, though. Let's use CSS to make it look a little better.

There was a time when Facebook did not allow linked style sheets. Thankfully, that isn't the case anymore. It did this to make it easier to filter your CSS. Facebook filters your style sheets⁷ to make sure you aren't changing the look and feel of anything outside your canvas area.

Let's start by moving our pagination links to the right side of the canvas area. We want to add the following style rules to our application:

Download `chapter5/karate_poke/public/stylesheets/application.css`

```
.pagination {
  float: right;
}
.battle {
  padding: 3px;
  font-size: large;
  clear: both;
}
.battle img {
  padding-right: 3px;
}
```

Now we just need to add a link to our layout:

```
<fb:fbml version="1.1">
  <%= stylesheet_link_tag "application"%>
  <%= fb_dashboard %>
  ...
```

When Facebook sees a linked CSS file, it will retrieve the page from your server and cache it indefinitely. There is no method you can call to update Facebook's cache. Instead, you will need to rename the file. Luckily, Rails handles this automatically. It appends each style sheet path with a timestamp parameter that changes whenever the file is changed, allowing Facebook to update its cache. You may need to restart your web server for the changes to be noticed.

7. Filtered CSS attributes are listed at http://wiki.developers.facebook.com/index.php/FBML#Invalid_CSS_attributes.

5.7 Summary

I know it has been a whirlwind tour, but you've now gotten a taste of what the Facebook canvas has to offer. We've built Karate Poke from an invitation screen and some models into a real application. Of course, there's still plenty more to explore. Next, let's take a look at some Facebook features we can use to make our application more interactive.

Chapter 6

Making It More Social

Now that we have attacks working, it's time to dig in to the Facebook social features. We'll start by notifying our users when they've been attacked using both normal Facebook notifications and email notifications. We'll look at how Facebook handles spam notifications and how to avoid getting your notifications turned off. Next, we'll use news feed items to publicize our users' actions. After that, we'll use discussion boards, comments, and walls to set up an area for discussion. Finally, we'll revisit some of our earliest code to make it better reflect the changes we've made so far.

All these social features act like free advertising for your application. They can help get your application in front of a wider audience.

6.1 Sending Notifications

We have our attack form working, but there's a problem. Our users won't know that they've been attacked until they visit their battles page. We need a way to tell people they've been attacked. We looked at using invitations to send messages in Section 2.2, *Creating the Invitation Form Using FBML*, on page 40, but that doesn't quite work here. Remember that invitations and requests require the user to approve the message before it can be sent. That would make our attack form a little harder to use. Additionally, the message is sent before our action gets called. That means we can't tell the message recipient if the attack was a hit or a miss.

Instead of using invitations, let's use Facebook notifications. Notifications appear at the top of a user's home page, as you can see in Figure 6.1, on the following page. Notifications are text-only messages that



Figure 6.1: New notifications appear on our home page.

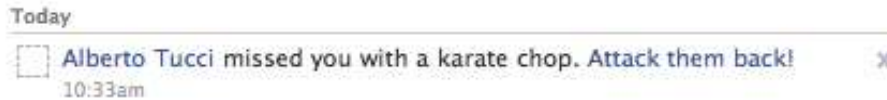


Figure 6.2: You can view your new notifications from the notifications tab of your inbox.

always start with the name of the sender. You can see a notification in Figure 6.2. Like invitations, notifications contain a message body that is specified in FBML.

Also like invitations, the number of notifications that your application can send on behalf of a user in one day is limited. Your application is dynamically assigned a maximum number of invitations based upon how many users ignore, hide, or report your notifications as spam.¹ If you attempt to exceed this limit, an exception will be raised.

There are a few differences between notifications and invitations. Even though both notifications and invitations use FBML for the body, the notification body is limited to text-only HTML and FBML tags. That means you can add a link to your notification, but not an image.

Sending a notification doesn't require user interaction. Because of this, Facebook allows you to see notifications that applications send on your behalf, as shown in Figure 6.3, on the following page. Facebook allows the sender to report a notification as spam or as inaccurate, as we will see in Section 6.1, *Spam Filtering*, on page 111.

1. This system is described in <http://developers.facebook.com/news.php?blog=1&story=77>.

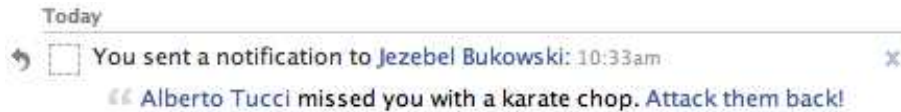


Figure 6.3: You can view notifications sent on your behalf.

Facebook Notifications

Now that we have a basic idea of what we can do with notifications, let's use them to notify the defender that they were just attacked. We'll use the `send_notification()` method on the `Facebooker::Session` object. The `send_notification` method takes just two parameters. The first is a list of user IDs to send the notification to. The second is the content of the notification.

Since we're notifying only the defender of the attack, our call is very simple. In the body, let's just include a message like the one that shows up on your battles page. Let's also include a link to our attack page so that the recipient can return the attack:

Download `chapter6/karate_poke/app/models/attack.rb`

```
def notify_defender
  message = <<-MESSAGE
  <fb:fbml>
  #{(hit? ? "hit" : "missed" ) }
  you with a #{move.name}.
  <a href="http://apps.facebook.com/karate_poke/attacks/new">
  Attack them back!</a>
  </fb:fbml>
  MESSAGE
  attacking_user.facebook_session.send_notification(
    [defending_user],message)
end
```

Now we can send users a notification when they've been attacked, but it doesn't feel very Railsy. Rails provides a much cleaner interface for sending emails in ActionMailer. Similarly, Facebooker provides the Publisher interface. Let's clean up our notification by converting it to use a Publisher.

First, we'll need to create a Publisher class. You can do this by running `script/generate publisher attack`. This will create `attack_publisher.rb` in `app/models` and a migration. Facebooker needs a table to track some information about our messages, as we will see in Section 6.2, *The*

Use Messaging as a Call to Action

If your application uses messaging, and it almost certainly should, make sure your messages encourage the recipient to take an action. If your message just says something like “Jen attacked you with a karate chop,” the recipient may just ignore the message. By adding a link that allows you to easily attack them back, you are encouraging the recipient to interact with both your application and their connections.

Part of developing a successful Facebook application is to keep your users engaged. After all, having a large number of installed users isn’t very interesting if nobody actually uses your application. You’ll see this concept in Karate Poke. Every time we send a message, we ask the recipient to perform some action.

Basics of Publishing Feeds, on page 113. Run `rake db:migrate` to create that table now.

Inside our `AttackPublisher`, we can define a method to send our notification. Because we can use the `Publisher` interface for sending more than just notifications, we’ll need to tell Facebooker that we want to send a notification.

We do this with the `send_as` method, as you can see here. We’ve included our `ApplicationHelper` module to allow us to use our helper methods here.

[Download](#) chapter6/karate_poke/app/models/attack_publisher.rb

```
helper :application
```

```
def attack_notification(attack)
  send_as :notification
  recipients attack.defending_user
  from attack.attacking_user.facebook_session.user
  fbml <<-MESSAGE
    <fb:fbml>
      #{attack_result(attack) }
      #{name attack.defending_user} with a #{attack.move.name}.
      #{link_to "Attack them back!", new_attack_url}
    </fb:fbml>
  MESSAGE
end
```

To create a notification, we need to specify a from user and one or more recipients. Since our publisher relies on the Facebook session of the sending user for sending our messages, our from user must be an instance of `Facebooker::User`. Our recipients can be an array of our `User` objects, a `Facebooker::User`, or even just a raw Facebook ID.

Using the `Publisher` interface gives us several advantages. Primarily, we have access to our view helpers including URL generation. This makes creating our notification body much easier. We also gain the ability to test our notifications without sending them. Just like with `ActionMailer`, we can call `AttackPublisher.create_attack_notification(attack)` to create a `Notification` object. If we want to both create the object and send it to Facebook, we can use `AttackPublisher.deliver_attack_notification(attack)`.

Now that we have our notification code cleaned up, let's set up our `Attack` model to automatically send a notification whenever an attack is created:

```
Download chapter6/karate_poke/app/models/attack.rb
```

```
after_create :send_attack_notification

def send_attack_notification
  AttackPublisher.deliver_attack_notification(self)
rescue Facebooker::Session::SessionExpired
  # We can't recover from this error, but
  # we don't want to show an error to our user
end
```

Let's give it a try. Use one of your test users to attack another test user. You should see the same notification in the defender's inbox and the attacker's sent notification area.

Sometimes you'll want to send a notification, but you don't have a good from user to use. For example, you could use a notification to update all your users about a new feature. In these cases, you can leave the from user blank in your publisher. If you don't specify a from user, `Facebooker` will send an announcement notification that appears to come directly from your application.

Notification via Email

Our notifications are working great, but the recipient has to log in to Facebook to know they have a message. This is fine for most notifications, but sometimes the notifications we send are more urgent. For example, if we were building a meetup application, we would want to let users know when an event was canceled. Facebook allows us to send notifications via email to any user who specifically opts in.



Figure 6.4: Facebook provides a simple framework for obtaining additional permissions.

Facebook provides a special FBML tag for requesting permission from a user. The `<fb:prompt-permission>` tag generates a special link that our users can click to give our application additional permissions. For example, to ask for the ability to send email, we can use this:

```
<%= fb_prompt_permission :email, "Can we email you from time to time?"%>
```

Facebook will render a link if a user has not already granted permission to our application. When clicked, the link opens a dialog box asking the user for their permission to send email, as shown in Figure 6.4. Once permission has been granted, the link will no longer be shown by Facebook.

Now that we have a way for users to opt in, we'll need a way to send email. To do that, we'll create a new action in our Publisher. Earlier, we used `send_as :notification` to tell Facebooker that we wanted to send a notification. Along with changing this to `:email`, we'll also specify a title for our message:

Download `chapter6/karate_poke/app/models/attack_publisher.rb`

```
def attack_notification_email(attack)
  send_as :email
  recipients attack.defending_user
  from attack.attacking_user.facebook_session.user
  title "You've been attacked!"
  fbml <<-MESSAGE
    <fb:fbml>
      #{attack_result(attack) }
      #{name attack.defending_user} with a #{attack.move.name}.
      #{link_to "Attack them back!", new_attack_url}
    </fb:fbml>
  MESSAGE
end
```



Figure 6.5: You can report a message as spam from your inbox.

Before you will be able to test our email notifications, you will need to add a permission prompt and then click the resulting link. Once you've done that, you can give our notifications a try from `script/console`.

Find the attack you just created, and use the publisher to call `attack_notification_email`. You should receive a notification email within a few minutes.

This feels like overkill for our notifications. Let's use the normal Facebook notifications instead of email notifications.

Spam Filtering

Facebook allows notification recipients to mark messages as spam, as you can see in Figure 6.5. To avoid being marked as spam, you should make sure your notifications are meaningful.

Notifications should be sent to users only when an action occurs that impacts them. We're probably safe because we send notifications only to people who are attacked. If we also sent notifications to other members of the defender's dojo, we might be marked as spammy.

You can see how spammy Facebook thinks your application is on the developer page. You can see the "Spamminess" metric in Figure 6.6, on the next page. If Facebook decides your application is spammy, you will lose the ability to send notifications for thirty days. After the thirty days is up, you can request that your application be unblocked and regain the ability to send notifications. Each additional blocking requires you to wait twice as long before you can request to be unblocked.



Figure 6.6: The Developer application shows your application's spamminess.

Even though our application sends notifications only to those people involved in the battle, we still might send too many notifications. We may want to send only one notification per battle per day. Let's look at how we could implement that.

First, we could add a `last_notified_on` column to our `User` model. Then, we could change our `send_attack_notification()` method to check this column.

```
def send_attack_notification
  if defending_user.last_notified_on != Date.today
    AttackPublisher.deliver_attack_notification(self)
    defending_user.update_attribute(:last_notified_on, Date.today)
  end
end
```

This is a simple way to reduce the number of notifications your application sends to its users.

The recipient isn't the only person who can report a notification as spam. The sender can also report a notification. When you view a sent notification, you can tell Facebook either that you didn't take the action mentioned in the notification or that you didn't want a notification to be sent.

Notifications are a great way of sharing information between two people. Next, we'll look at ways of spreading information among a group of people.



Figure 6.7: One-line and short feed items are shown here.

6.2 Publishing to News Feeds

Notifications are a great way to tell a user about an action that affects them. Feed items, on the other hand, are meant as a way of sharing information about a user's actions. You've probably seen Facebook feed items already.

They are the little messages that show up on your home page and in your mini-feed. They have a title, a body, and optionally images. Feed items come in three sizes: one-line stories, short stories, and full-size stories. One-line and short stories are shown in Figure 6.7. Our users can increase the size of a story by clicking the edit button to the right of the feed item, as shown in Figure 6.8, on the next page.

Along with having three different sizes of feed items, Facebook provides two different methods for creating feed items. We'll start by looking at how we can create feed items from inside our application. Once we have a firm grip on the basics of feed items, we'll learn how to use a *profile publisher* to allow our users to create their own.

The Basics of Publishing Feeds

The process of creating feed items has evolved more than any other part of the Facebook Platform. Facebook released its third version of the feed publishing API in July 2008. Publishing feed items is now a two-step process. The first part involves creating and registering a template that your feed items will follow. Once you have created and registered a template, you can use this template to create feed items.

This sounds more complicated than it really is. Feed templates are just a collection of simple strings that contain variables. If we wanted to create a feed item that said "Jen hit Mike with a karate chop," then we might create a simple template that looks like `{*actor*} {*result*}`



Figure 6.8: Facebook allows users to change the size of their feed items.

{*defender*} with a {*move*}. Once you have created and registered a template (we'll see how to do this in a moment), publishing a feed becomes as simple as specifying what values to use for each variable.

Let's look at what it takes to create a simple template. We'll start by adding a new method to our publisher. This method will be responsible for creating our template. Unlike our previous publisher methods, it won't be used for sending a message. Facebooker requires that methods used for generating templates be suffixed with `_template`. Since we're publishing a feed about an attack, let's call our new method `attack_feed_template`. Our method will do only one thing. It will create a one-line template like the earlier one.

```
def attack_feed_template
  one_line_story_template "{*actor*} {*result*} {*defender*}" +
    " with a {*move*}."
end
```

Notice that our template starts with `{*actor*}`. Facebook requires that all templates start with this variable, which represents the user who performed the action. Now that we have a method that creates a template, we'll need to register our template with Facebook. We'll do this by calling `AttackPublisher.register_attack_feed`. Give this a try in script/console. You should get an integer as a return value.

The integer returned by the call to register is the `template_id`. We'll need to send this identifier to Facebook when we want to publish an instance of this template. Don't worry about writing down the number; Facebooker will deal with remembering this for you. Template IDs are stored in the `facebook_templates` table that was created automatically for us when we generated our first publisher.

Now that we have a registered template, let's actually publish a feed item. We'll need to create another method to do this. Our new method is responsible for providing values for all the variables in our templates. Its name should match the name of our template method without the `_template` suffix.

```

def attack_feed(attack)
  send_as :user_action
  from attack.attacking_user.facebook_session.user
  data :result=>attack_result(attack),
       :move=>attack.move.name,
       :defender=>name(attack.defending_user)
end

```

end

As with notifications, we'll need to tell Facebooker what kind of message we want to send. In this case, we use `send_as :user_action`. Also like in our notifications, we provide the `Facebooker::User` for whom we want to send our message. Next, we use the `data` method to provide all our template variables except for `actor`. Facebook knows to use the user who is publishing as the actor in our template. We now have everything we need to actually publish a feed. Try it out by calling `AttackPublisher.deliver_attack_feed(an_attack)`. Find an attack in `script/console`, and try it. You should see a new item on the profile of the sending user.

Earlier, I mentioned that there are multiple sizes of stories. In the previous example, we created only a one-line template. Let's add a short story template as well:

```

def attack_feed_template
  one_line_story_template "{*actor*} {*result*} {*defender*}" +
    " with a {*move*}."
  short_story_template "{*actor*} engaged in battle.",
    "{*actor*} {*result*} {*defender*} with a {*move*}."
end

```

end

Now, our user will be able to decide whether they want only a one-line story in their feed or whether they want a slightly larger story. By default, all stories are published as one-line stories. A user has to explicitly edit their feed to increase the size of a story. Let's see what our new item looks like. Reregister this template in `script/console`, and then publish another feed item.

After publishing a new item, go to your profile. You should see your new feed item. Click Edit, and you should see the option to change the size of this feed item. If you don't see this option, you may have forgotten to reregister your template. We'll have to reregister our template every time we make a change to it.

Along with providing a title and a body, Facebook also allows us to include images with our short story. If we add an `images` parameter to the data we send to Facebook, it will include our images in short stories. Let's see what that looks like:

```
def attack_feed(attack)
  send_as :user_action
  from attack.attacking_user.facebook_session.user
  data :result=>attack_result(attack),
       :move=>attack.move.name,
       :defender=>name(attack.defending_user)
       :images=>[image(attack.move.image_name,new_attack_ur)]
end
```

The Facebooker image helper lets us just specify a filename and a URL to which the image should be linked. It takes care of setting up the right Facebook parameters. Try publishing another story. You won't need to reregister your template because it hasn't changed. We're just adding more data.

We've looked at two of the three sizes of feed items. Full-size story templates are created just like short-story templates. The only difference is that the full size story can be up to 700 pixels tall. That's much more room than we can possibly use for our simple application. We'll just stick to our two existing sizes.

Feed Item Aggregation

Feed items show up in two different places. Your actions show up on the wall tab of your profile page. Your friends' actions show up on your home page. Although every feed item that is created on your behalf shows up on your profile, not every one of your friends' actions shows up on your home page. Because of the overwhelming number of feed items created, Facebook tries to group similar feed items to reduce the number of redundant feed items.

Let's take a look at an example. Let's say you are friends with both me and my wife on Facebook. If we attacked each other back and forth several times, it would generate a large number of feed items. Instead of showing you a play-by-play our attacks, Facebook would rather show you a single story that said something like "Mike and Jen attacked their friends with Karate Poke."

Facebook's most recent changes to the feed API are aimed at better enabling exactly this kind of aggregation. Along with providing multiple sizes of templates, Facebook also allows you to provide multiple one-line and short-story templates. Because Facebook can combine items only when all the variables match, each new version should contain progressively less information. Facebook can then use these more generic templates for combining feed items. If Facebook can aggregate

our story, it is much more likely to show up in our friends' news feeds. That means more exposure for our application.

Let's now add versions of our story templates that can more easily be aggregated:

```
def attack_feed_template
  attack_back=link_to("Join the Battle!",new_attack_url)
  one_line_story_template "{*actor*} {*result*} {*defender*}
    with a {*move*}. #{attack_back}"
  one_line_story_template "{*actor*} are doing battle
    using Karate Poke. #{attack_back}"
  short_story_template "{*actor*} engaged in battle.",
    "{*actor*} {*result*} {*defender*} with a {*move*}. #{attack_back}"
  short_story_template "{*actor*} are doing battle using Karate Poke.",
    attack_back
end
```

Now we can reregister our templates and send a few feed items. If you're lucky, you may see an aggregated feed from your application show up in your news feed. Unfortunately, there is no way to guarantee that a published feed will be aggregated. This can make testing difficult. Facebook does provide a tool for testing feed aggregation.² Did you notice that our wording in our more generic feed examples assumed that the actor parameter would include multiple people? The only time Facebook will use a story other than the first one is when multiple actors are involved.

Making Our Application More Visible

We just talked about how aggregated stories are more likely to be shown in your friends' news feeds. Another way to increase the likelihood that your story will be shown is to have the links in your story point to pages viewable by a user without authentication. When we configured our Rails application, we used the `ensure_authenticated_to_facebook` filter to require all users to log in to our application. Let's remove that restriction from our battles page to make it publicly viewable. To do that, we need to disable the `ensure_authenticated_to_facebook` filter:

```
skip_before_filter :ensure_authenticated_to_facebook,
  :only=>:index
```

2. Available at <http://developers.facebook.com/tools.php?feed>. Unfortunately, this tool hasn't been updated to reflect the new API as of July 2008.

Since Facebook does not send us session information for users who haven't added our application, we'll also need to modify our `set_current_user` method:

```
def set_current_user
  set_facebook_session
  # if the session isn't secured, we don't have a good user id
  if facebook_session and facebook_session.secured?
    self.current_user =
      User.for(facebook_session.user.to_i, facebook_session)
  end
end
```

That's almost enough to make the battles page visible to a user who hasn't added our application. When a non-logged-in user visits our application, `current_user` will return `nil`. When this happens, our attack form will call `available_moves` on the `nil` `current_user`. This will raise an exception. We can fix that by showing the attack form only if `current_user` is not `nil`. If it is, let's show the user a page that encourages them to log in to Karate Poke.

To accomplish this, we'll need to modify both the controller and the view. Let's start with the controller. The current user will be unknown in two cases. The first is when the user clicks a message and the `user_id` parameter is given to us. Our code works in this case. The second is when a new user visits our application for the first time. When this happens, we want the user to have to authorize our application. We can just redirect the user to the authorization page and return to stop execution, as shown here:

```
def index
  if params[:user_id]
    @user = User.find(params[:user_id])
  else
    @user = current_user
  end
  # If we don't have a user, require add
  if @user.blank?
    ensure_authenticated_to_facebook
    return
  end
  ...
end
```

Along with changing the controller, we will also need to fix our view. Our view will need to change to remove the attack form for non-logged-in users. We can replace it with a simple message. To do this, we can check to see whether the current user is blank. We can't use an FBML

if tag here. The FBML if tag would still try to render the form, which would raise an exception when we tried to get the available moves for a nil user.

```
<% if current_user.blank? %>
<h3><%=link_to "Add Karate Poke",new_attack_path%>
  to attack <%=name @user%></h3>
<% else %>
  <h3>Do you want to attack <%= name @user%>?</h3>
  <% facebook_form_for Attack.new do |f| %>
    <%= f.collection_select :move_id, current_user.available_moves,
      :id, :name, :label=>"Move" %> <br />
    <%= hidden_field_tag "ids[]", @user.facebook_id%>
    <%= f.buttons "Attack!" %>
  <% end %>
<% end %>
```

Now our battles page is visible to all Facebook users. Our feeds are also more likely to be shown. There's still more we can do.

The Facebook Profile Publisher

So far, we've looked only at messages generated programmatically from within our application. With the Facebook Platform update released in July 2008, Facebook introduced the Facebook Profile Publisher as a way to allow users to create their own feed items. The Facebook Profile Publisher, shown in Figure 6.9, on the following page, places your application on the profiles of your users and their friends. Even though they are similarly named, the Facebooker Publisher and the Facebook Profile Publisher are very different. The Facebooker Publisher is used for sending messages to Facebook. The Profile Publisher is an interface that allows users to create feed entries from a profile page.

The Profile Publisher isn't meant to be a general-purpose application area. It has a very specific focus—adding content to the profiles of our users. The interaction pattern is simple. When a user selects your application from the pull-down on the right of Figure 6.9, on the next page, Facebook fetches a single form from our server and places it in the profile area. The user fills out the form, and Facebook submits it to our application. At that point, we can return a newly created feed item, or we can return an error. That's the extent of the interaction.

For Karate Poke, this simple interaction is enough for us to build our attack form. First, we'll need a controller. We could add this to an existing controller, but the interaction is different enough that it makes sense to create a new one. Let's generate a new controller by running



Figure 6.9: The Profile Publisher makes it easy to add content to your friends' feeds.

script/generate controller profile_publisher. We won't need to add a route for our controller, because the Rails default routes will take care of it.

Now that we have a controller, let's implement just enough to make our Profile Publisher show up when we visit our friends' profiles. Facebook has a somewhat complicated API for creating a publisher form.³ Thankfully, Facebooker has hidden most of that from us. To have Facebook display a form, we'll just need to call the `render_publisher_interface` controller method. It takes a string containing the content to display as its only required parameter. We can get our form as a string using `render_to_string`. We'll also need to know which user is being attacked. Facebook will send us the Facebook ID of that user in the `fb_sig_profile_user` parameter. That gives us a very simple method:

```
def index
  @defender = User.for(params[:fb_sig_profile_user])
  render_publisher_interface(render_to_string(:partial=>"form"))
end
```

With our basic controller in place, we need a form. Our attack form

3. You can view the details at http://wiki.developers.facebook.com/index.php/New_Design_Publisher.

Figure 6.10: The Profile Publisher is configured in the Developer application.

is basically what we need. Unfortunately, we can't use the same code as we did on our attack form. The Profile Publisher doesn't want us to include `<form>` tags in our code.

`Download` chapter6/karate_poke/app/views/profile_publisher/_form.erb

```
How do you want to attack <%=name(@defender)%>?<br />
<%= collection_select :attack,:move_id,
  current_user.available_moves, :id, :name%>
```

The coding part of our basic form is done. Now we just need to do a little configuration. In the Developer application is a section for publishing content to friends. We'll want to add the URL of our index action in the Callback URL portion of the form. The string you put in the Publishing Action field will be displayed in the user's profile. You can see what this looks like in Figure 6.10.

The Profile Publisher has two different configuration areas. The first, Publish Content to Friend, controls what is seen when you view the profile of your friends or when they view your profile. In our case, we want to show our attack form. The second area, Publish Content to Self, determines what shows when you view your own profile. In our case, we will leave this area blank to keep our users from attacking themselves.

With our configuration done, hit Save, and go to the profile of one of your friends. You should now have the option of attacking them from their profile. You can give it a try, but it won't do anything yet. Before it will work, we'll need to code the form submission process.

When a user clicks the Post button on our Profile Publisher, Facebook will send another request to the URL we specified in our configuration. Our form parameters will be included, but not where you might expect. Our form contained only the `move_id` of the attack. Normally, we would be able to create an attack using `Attack.new(params[:attack])`. Inside the

Profile Publisher, however, Facebook adds another level to our parameters. Instead, we'll need to use `Attack.new(params[:app_params][:attack])`.

Now that we know where to find our form parameters, we can go about creating our attack. Once we've created our attack, we'll need to give Facebook a feed item to be displayed in the user's feed. Earlier, we created an `attack_feed` method in our `AttackPublisher`.

We can use the Facebooker `render_publisher_response` method to send this feed item back to Facebook. That means our code to process the form submission looks like this:

```
@defender = User.for(params[:fb_sig_profile_user])

attack = Attack.new(params[:app_params][:attack])
@attack = current_user.attack(@defender, attack.move)
render_publisher_response(AttackPublisher.create_attack_feed(@attack))
```

Since Facebook sends our form parameters to the same URL it sends the form request to, we'll need a way of determining what Facebook is looking for. Facebooker provides the `wants_interface?` method to tell us whether Facebook wants the Profile Publisher interface or whether it wants us to process the form. We can combine our code for displaying a form and our code for processing a form to get our almost final index action:

```
def index
  @defender = User.for(params[:fb_sig_profile_user])
  if wants_interface?
    render_publisher_interface(render_to_string(:partial=>"form"))
  else
    attack = Attack.new(params[:app_params][:attack])
    @attack = current_user.attack(@defender, attack.move)
    render_publisher_response(
      AttackPublisher.create_attack_feed(@attack))
  end
end
```

If we get an error during form processing, Facebook gives us a way to provide a helpful message to the user. For example, if we had a more complex form that required validation, we would want to let the user know about a validation failure. We would do this using the `render_publisher_error` method. This method takes two parameters, a title line and a body line. Thankfully, our form is simple enough that we don't have to worry about error handling.

We're almost done with our Profile Publisher implementation. We've covered rendering the form and handling data from our users. There is

just one more case to handle. When a nonuser of our application views the profile of one of our application's users, the nonapplication user will be able to use our Profile Publisher. When this happens, Facebook will provide our application with their Facebook ID but won't provide a session key. Currently, our `set_current_user` method won't set the `current_user` if no session is provided. Since this is the only case where we want a user without a session, we can add code directly to our index action to handle these users:

Download chapter6/karate_poke/app/controllers/profile_publisher_controller.rb

```
class ProfilePublisherController < ApplicationController
  skip_before_filter :ensure_authenticated_to_facebook
  def index
    if current_user.nil? and facebook_params[:user]
      self.current_user = User.for(facebook_params[:user])
    end

    @defender = User.for(params[:fb_sig_profile_user])
    if wants_interface?
      render_publisher_interface(render_to_string(:partial=>"form"))
    else
      attack = Attack.new(params[:app_params][:attack])
      @attack = current_user.attack(@defender, attack.move)
      render_publisher_response(
        AttackPublisher.create_attack_feed(@attack))
    end
  end
end
```

That takes care of setting up the user for our action. There is one more problem where the lack of session will cause us problems. When an attack is created, we send a notification from the attacker to the defender. Since our attacker won't have a session in this case, we'll need to handle that error. An easy fix is just to rescue the exception that is raised, as shown here:

Download chapter6/karate_poke/app/models/attack.rb

```
after_create :send_attack_notification

def send_attack_notification
  AttackPublisher.deliver_attack_notification(self)
rescue Facebooker::Session::SessionExpired
  # We can't recover from this error, but
  # we don't want to show an error to our user
end
```

With that done, users and nonusers alike can use our Profile Publisher

to attack their friends. In less than fifty lines of code we've created a simple way to encourage our users to interact with our application. Next, we'll look at another way to encourage interaction by building a comment area.

6.3 Comments and Discussion Boards

We've looked at a lot of social features, but none of them has involved multiple people interacting. Since physical sports often involve verbal sparring, let's give our users a place for that. We'll look at a few different implementations of this concept. We'll focus our attention on our users' battles pages, although the same concept could be used anywhere you want people to be able to interact.

Adding a Comment Area

We're going to start building our comment area with the view. This will allow us to figure out what models we'll need to build. Facebook provides two tags for building walls. They are `<fb:wall>` and `<fb:wallpost>`. As you might expect, Facebooker provides the `fb_wall()` and `fb_wallpost()` helpers.

Wall posts require only the ID of the poster and the body of the comment to display. Even though we haven't implemented the model or the controller, let's sketch out a view for our comment wall:

```
<% fb_wall do %>
  <% for comment in @comments %>
    <%= fb_wallpost comment.poster, comment.body %>
  <% end %>
<% end %>
```

For this to work, we'll need to build the Comment model. Our view suggests to me that each comment will need to have at least the ID of the poster and the body of the comment. We'll also need to associate our comment with whatever wall we want it to display on. Instead of creating a Wall model, let's just associate the comments with the user who is being commented on. Since we're going to be looking up our comments based upon the `user_id` and ordering them based upon when they are created, let's create an index on those fields:

[Download](#) chapter6/karate_poke/db/migrate/010_create_comments.rb

```
class CreateComments < ActiveRecord::Migration
  def self.up
    create_table :comments do |t|
```

```

      t.integer :user_id
      t.integer :poster_id
      t.text :body
      t.timestamps
    end
    add_index :comments, [:user_id,:created_at]
  end

  def self.down
    drop_table :comments
    remove_index :comments, [:user_id,:created_at]
  end
end

```

Now we just need to add a couple of associations to our User and Comment models:

[Download](#) chapter6/karate_poke/app/models/user.rb

```

has_many :comments
has_many :made_comments, :class_name=>"Comment", :foreign_key=>:poster_id

```

[Download](#) chapter6/karate_poke/app/models/comment.rb

```

class Comment < ActiveRecord::Base
  belongs_to :user
  belongs_to :poster, :class_name=>"User"
end

```

Let's also add a `comment_on()` method to the User model to encapsulate the logic of creating a comment:

[Download](#) chapter6/karate_poke/app/models/user.rb

```

def comment_on(user,body)
  made_comments.create!(:user=>user,:body=>body)
end

```

Let's create a comments controller with a create action. The create action will create the comment and then redirect the user to the battles page to which the comment was added. We'll also want to add the comments resource:

[Download](#) chapter6/karate_poke/app/controllers/comments_controller.rb

```

class CommentsController < ApplicationController
  def create
    comment_receiver = User.find(params[:comment_receiver])
    current_user.comment_on(comment_receiver,params[:body])
    redirect_to battles_path(:user_id=>comment_receiver.id)
  end
end

```

```
Download chapter6/karate_poke/config/routes.rb
```

```
map.resources :comments
```

Now that we have our controller, we need to create a form that includes two parameters, `comment_receiver` and `body`. We want to display this on our battles page, so let us add it to the bottom of `app/views/attacks/index.fbml.erb`. Our form should be subtle, so we're going to use a normal HTML form instead of a Facebook-specific form:

```
<% form_for Comment.new do %>
  Talk some trash: <br />
  <%= text_area_tag :body %> <br />
  <%= hidden_field_tag :comment_receiver,@user.id %>
  <%= submit_tag 'Post' %>
<% end %>
```

Finally, we need to add the remainder of our view to `index.fbml.erb`. That uses a list of comments in `@comments`, so we should create that variable in the index action:

```
Download chapter6/karate_poke/app/controllers/attacks_controller.rb
```

```
if @battles.blank?
  flash[:notice]="You haven't battled anyone yet."+
    " Why don't you attack your friends?"
  redirect_to new_attack_path
else
  @comments = @user.comments
end
```

You should be able to post comments now. There is one more thing we need to do before we call this feature done. Because we're displaying content directly from our users, we need to sanitize the content. As it stands now, a user could add either HTML or FBML to our pages. Fixing this hole is simple; we can use the rails `h()` helper to strip the tags from our comments:

```
<%= fb_wallpost comment.poster, h(comment.body) %>
```

That's it. In less than fifty lines of code, we've added a nice comment system to our application.

Using the Built-in Comments

Of course, fifty lines of code isn't as nice as a single line of code. Thanks to the `<fb:comments>` FBML tag, we can eliminate all our code. The `<fb:comments>` tag adds a comment wall to any page.

The `<fb:comments>` tag takes quite a few parameters.⁴ There are four required parameters: `xid`, `canpost`, `candelele`, and `numposts`. The first parameter, `xid`, is a unique name given to this wall. You could place the same wall on multiple pages by specifying the same value for `xid` to each `<fb:comments>` tag. Since we want a different comment wall on each user's page, we'll use the string `User_` followed by the user's id as the `xid` of our wall.

The next two parameters, `canpost` and `candelele`, tell whether the current user can post on the wall and delete posts, respectively. We want to allow everybody to post, but only the owner of the battles page should be able to delete posts.

The final parameter tells how many posts to show for this wall. Let's show the ten most recent posts. Facebook will render a "view all" link if there are more than the requested number of posts.

We'll also specify one optional parameter, `showform`, to tell Facebook to include the new comment form inside the wall. If we don't specify `showform`, users will have to go to another page to post a comment.

```
<%= fb_comments "User_#{@user.id}", true,
  current_user==@user,10, :showform=>true %>
```

There is one gotcha with the `<fb:comments>` tag. During the comment-posting process, Facebook will fetch your page several times to retrieve configuration information from the `<fb:comments>` tag. Because they make these requests as POSTs, you may run into problems if you're using RESTful resources. As a workaround, we need to create a route to any page with comments on it that will accept a post. For example, we could create the following route:

```
map.comments 'battles/:user_id/comments',
  :controller => 'attacks', :action => 'index'
```

and then add the `callbackurl` parameter on our comments. Adding the `callbackurl` parameter will make sure that Facebook can fetch our battles page whether they use a GET or a POST request:

```
<%= fb_comments "User_#{@user.id}", true,
  current_user==@user,10, :showform=>true,
  :callbackurl => comments_url(:user_id=>@user.id) %>
```

Take a look at that. It looks a little better than the wall we built, and it took only one line of code. Of course, there are several drawbacks. When comments are posted to our wall, they are not sent to

4. You can see them all at <http://wiki.developers.facebook.com/index.php/Fb:comments>.



Figure 6.11: The `<fb:board>` tag provides a one-line discussion forum.

our application. We also can't access them using the Facebook API. Facebook doesn't give us any method of accessing the comments that are posted on our pages. This means that we are locked in to using the `<fb:comments>` tag. It also means we can't make our comments searchable. Since we've already implemented our comment functionality, let's use that in place of the `<fb:comments>` tag.

Discussion Board

Facebook provides more than just a single, unthreaded comment wall. It also provides the `<fb:board>` tag for including an entire message board on a page. It has a few more options than `<fb:comments>`, but it works similarly. Implementing comments on our own was easy, but a full discussion board would be much more work. Let's give it a try on our battles page:

```
<%= fb_board "User_#{@user.id}" %>
```

You can see what this looks like in Figure 6.11.

6.4 Spreading by Invitation

We started our application by building invitation functionality. We built something that works, but it doesn't fit in well with our application. Let's revisit the invitation process to add some polish.

We started with a very basic invitation page:

```
<fb:add-section-button section="profile" />
<% fb_multi_friend_request("Karate Poke",
  "Invite your friends to use Karate Poke.",
  invitations_path) do %>
  Attack your friends. Install Karate Poke now.
  <%= fb_req_choice("Attack!",
    new_invitation_path(:from=>facebook_session.user.to_s))%>
<% end %>
```


Now that we have a more complete application, how about making a few changes? Let's send new users to the new attack page instead of the new invitation page. While we're at it, we can change the wording of our invitation form to better match our application:

```
<% fb_multi_friend_request("Karate Poke",
  "Invite your friends to join your Karate Poke Dojo.",
  invitations_path) do %>
  Come join my Karate Poke Dojo and fight your friends.
  <%= fb_req_choice("Join the battle!",
    new_attack_path(:from=>@user.to_s,:canvas=>true))%>
<% end %>
```

That's a step in the right direction, but there's still more we can do. We currently allow our users to send invitations to people who are already users of the application. Let's show only those friends who don't have a sensei. We built the potential disciples method earlier. Let's use that to exclude users who already belong to a dojo.

Facebook lets us specify a list of IDs to exclude from our friend picker. We'll create a `@not_potential_disciples` instance variable to specify which users to exclude:

```
friend_ids = params[:fb_sig_friends].split(/,/,)
@not_potential_disciples =
  friend_ids - current_user.potential_disciples(friend_ids)
```

We have a problem. Facebooker doesn't provide a way to pass options to the multifriend input. We'll need to change the way we build our invitation view to fix this.

To get more control over the request, we can replace our `fb_multi_friend_request` with an `fb_request_form`. The `fb_request_form` uses its `block` parameter to hold the content of the form.

To specify the content for our invitation, we'll need to use a `content_for` block:⁵

Download chapter6/karate_poke/app/views/invitations/new.erb

```
<% content_for("invite_message") do %>
  Come join my Karate Poke Dojo and fight your friends.
  <%= fb_req_choice("Join the battle!",
    new_attack_path(:from=>current_user,:canvas=>true))%>
<% end %>
<% fb_request_form("Karate Poke","invite_message",
  invitations_path) do %>
```

5. The `content_for` method is one of the most underutilized helpers in Rails. It allows a portion of a view to be stored under a string key and retrieved later. It is documented at <http://api.rubyonrails.com/classes/ActionView/Helpers/CaptureHelper.html#M001748>.

```

<%= fb_multi_friend_selector(
  "Invite your friends to join your Karate Poke Dojo.",
  :exclude_ids=>@not_potential_disciples.join(",")%>
<br />
<%= fb_request_form_submit %>
<% end %>

```

There's just one more thing to do. When a user clicks the Skip button in our invitation, they currently receive an error. This happens because Facebook sends a GET request to the URL we specify in our invitation form. The result is a request to our nonexistent index action. Since Facebook doesn't give us a way to change where the Skip button links to, we will need create an index action that simply redirects the user to their battles page:

Download chapter6/karate_poke/app/controllers/invitations_controller.rb

```

def index
  redirect_to battles_path
end

```

That takes care of the invitation. Now we just need to correctly set the sensei of our users when they join:

Download chapter6/karate_poke/app/controllers/attacks_controller.rb

```

def new
  if params[:from]
    current_user.update_attribute(:sensei, User.find(params[:from]))
  end
end

```

Finally, let's add our `<fb:add-section-button>` tag to our layout to make sure our users will see it. While we're at it, we can replace the raw FBML with a Facebooker helper. You can see that change here:

Download chapter6/karate_poke/app/views/layouts/application.fbml.erb

```

<fb:fbml version="1.1">
<%= stylesheet_link_tag "application"%>
<%= fb_dashboard %>

<% fb_tabs do %>
  <%= fb_tab_item "Your Battles", battles_path, :selected=>!@battles.nil? %>
  <%= fb_tab_item "Attack!", new_attack_path, :selected=>@battles.nil? %>
  <%= fb_tab_item "Help & Feedback", fb_about_url, :align=>"right"%>

<% end %>
<br />
<%= fb_add_profile_section %>
<%= yield%>
</fb:fbml>

```

That makes our invitations feel more like part of Karate Poke. We're building dojos and encouraging our users to spread our application.

6.5 Giving the Profile a Makeover

When we last looked at the profile, we had just started working on Karate Poke. Since then, we've built quite a lot of functionality. What we put into our users' profiles is looking a little dated. Let's replace it with something that better reflects our application, like the user's recent battle history.

Updating Profiles with a Publisher

We will start by revisiting how we update our users' profiles. Earlier, we looked at setting the profile FBML directly using the `profile_fbml=` method. This worked for our purposes, but it required us to build our FBML by hand. Instead, we'll turn to the Facebooker Publisher for help.

Let's start by looking at the easiest possible profile update we can do with the publisher. Add the following example to your `AttackPublisher`, and run it by calling `AttackPublisher.deliver_profile_update`:

```
def profile_update(user)
  send_as :profile
  recipients user
  profile "This is a test"
end
```

You can run that, and it works, but it's certainly not what we want our final profile to look like. Our final profile view should probably look something like the attack list on our battles page. In fact, let's start by creating a new partial and reusing that code. Since our partial is for our `AttackPublisher`, I'll put it in `app/views/attack_publisher/_profile.erb`:

```
<fb:fbml>
<% for attack in @battles %>
  <div class="battle">
    <%= image_tag attack.move.image_name %>
    <%= link_to(
      name(attack.attacking_user, :linked=>false),
      battles_path(:user_id=>attack.attacking_user)) %>
    <%= attack_result(attack) %>
    <%= link_to(
      name(attack.defending_user, :linked=>false),
      battles_path(:user_id=>attack.defending_user)) %>
    with a <%= attack.move.name %>
  </div>
<% end %>
</fb:fbml>
```

To make that work, we'll just have to do two things. First, we'll need to retrieve a list of battles for our user. After that, we'll need to tell the publisher to render our new profile partial. Both of these will take just a few lines of code:

```
def profile_update(user)
  send_as :profile
  recipients user.facebook_session.user
  @battles=user.battles
  profile render(:partial=>"profile",:assigns=>{:battles=>@battles})
end
```

Isn't that easy? The main thing to be aware of is that you need to explicitly pass variables into your template using `:assigns => {}`. The variables that you pass in will be defined as instance (or `@`) variables in your template. With that done, we'll just need to call the `deliver_profile_update` method after we create an attack. We can do that with an `after_create` callback like the one shown here:

```
after_create :update_profiles

def update_profiles
  AttackPublisher.deliver_profile_update(attacking_user)
  AttackPublisher.deliver_profile_update(defending_user)
end
```

Give it a try, and look at the results. There are a couple of important things to notice. First, images in your users' profiles aren't being loaded from your server. Facebook caches all profile images to ensure that profile pages load quickly. When you set profile FBML, Facebook will find all the images in your FBML and cache them. Facebook will refresh their image cache only if the name of an image changes. Rails' `image_tag` helper includes a last-modified time as part of the URL, so we shouldn't have to worry about this.

Along with the cached images, you'll also notice that our links are broken. So far, our entire application has lived inside the Facebook canvas. We've been able to use relative links because our entire application has lived at `apps.facebook.com`. Profiles, on the other hand, use the `www.facebook.com` hostname. We'll have to make sure all our links on our users' profiles point to the right place.

Currently, we use the `battles_path` helper to generate our links. That method will generate only the path portion of our link. To include the correct hostname, we'll need to change our view to use the `battles_url` helper instead. (This one little change is responsible for at least 50

percent of the time I spend debugging profile problems. I've started adding a note to my monitor reminding me to make all my links use the `_url` helpers.) Once we convert our helpers, the Facebooker publisher will make sure they point to the canvas page. You can see the changed code here:

```
<fb:fbml>
<% for attack in @battles %>
  <div class="battle">
    <%= image_tag attack.move.image_name %>
    <%= link_to(
      name(attack.attacking_user, :linked=>false),
      battles_url(:user_id=>attack.attacking_user)) %>
    <%= attack_result(attack) %>
    <%= link_to(
      name(attack.defending_user, :linked=>false),
      battles_url(:user_id=>attack.defending_user)) %>
    with a <%= attack.move.name %>
  </div>
<% end %>
</fb:fbml>
```

That should fix our broken links. Create another attack, and double-check your updated profile.

The Skinny and Fat on Profiles

Our profile content shows up in the wide part of our users' profiles by default. You can change this setting in the Developer application. In the wide area, our content can be up to 380 pixels wide. In the narrow view, our content can be only 180 pixels wide. Since our users can move their profile box from one side of their profile to the other, we should make sure our content looks good on both sides.

Since the two sections of the profile have such drastically different widths, Facebook gives us the `<fb:wide>` and `<fb:narrow>` FBML tags to specify markup specific to each area. Our existing profile box looks fine on the wide side, so we can wrap our existing code in an `<fb:wide>` tag. We can add an `<fb:narrow>` tag immediately afterward that shows just our users' first names and eliminates some of the descriptive text:

```
<fb:fbml>
<fb:wide>
  <% for attack in @battles %>
    <div class="battle">
      <%= image_tag attack.move.image_name %>
      <%= link_to(
        name(attack.attacking_user, :linked=>"false"),
```

```

        battles_url(:user_id=>attack.attacking_user)) %>
      <%= attack_result(attack) %>
      <%= link_to(
        name(attack.defending_user, :linked=>"false"),
        battles_url(:user_id=>attack.defending_user)) %>
      with a <%= attack.move.name %>
    </div>
  <% end %>
</fb:wide>
<fb:narrow>
  <% for attack in @battles %>
    <div class="battle">
      <%= image_tag attack.move.image_name %>
      <%= link_to(
        name(attack.attacking_user, :linked=>"false",
          :firstnameonly=>true),
        battles_url(:user_id=>attack.attacking_user)) %>
      <%= attack_result(attack) %>
      <%= link_to(
        name(attack.defending_user, :linked=>"false",
          :firstnameonly=>true),
        battles_url(:user_id=>attack.defending_user)) %>
    </div>
  <% end %>
</fb:narrow>
</fb:fbml>

```

Our narrow view looks better, but there's one more part of the profile that we haven't dealt with. Our users can choose to move our profile box from the boxes tab to their main profile area using the edit menu shown in Figure 6.12, on the following page.

The main profile box is the same width as the narrow box but is limited in height to only 250 pixels. Thankfully, our narrow profile view is already relatively short. If we just limit our main view to the first four attacks, we should have plenty of space. We can move our narrow profile code to a partial and use that for both the narrow box and the main profile area. Once we have our partial created, we can update our `profile_update` method as well.

OK, let's start with the partial extraction. We can create a new partial called `_profile_narrow.erb`:

Download chapter6/karate_poke/app/views/attack_publisher/_profile_narrow.erb

```

<% for attack in @battles %>
  <div class="battle">
    <%= image_tag attack.move.image_name %>
    <%= link_to(

```



Figure 6.12: You can easily move a profile box around.

```

name(attack.attacking_user,:linked=>"false",
    :firstnameonly=>true),
battles_url(:user_id=>attack.attacking_user)) %>
<%= attack_result(attack) %>
<%= link_to(
    name(attack.defending_user,:linked=>"false",
        :firstnameonly=>true),
    battles_url(:user_id=>attack.defending_user)) %>
</div>
<% end %>

```

Then we can modify our `_profile.erb` template to use our new narrow template:

```
Download chapter6/karate_poke/app/views/attack_publisher/_profile.erb
<fb:fbml>
<fb:wide>
  <% for attack in @battles %>
    <div class="battle">
      <%= image_tag attack.move.image_name %>
      <%= link_to(
        name(attack.attacking_user, :linked=>"false"),
        battles_url(:user_id=>attack.attacking_user)) %>
      <%= attack_result(attack) %>
      <%= link_to(
        name(attack.defending_user, :linked=>"false"),
        battles_url(:user_id=>attack.defending_user)) %>
      with a <%= attack.move.name %>
    </div>
  <% end %>
</fb:wide>
<fb:narrow>
  <%= render :partial=>"profile_narrow"%>
</fb:narrow>
</fb:fbml>
```

With that done, we can add a call to `profile_main` to set the FBML for our main profile box:

```
Download chapter6/karate_poke/app/models/attack_publisher.rb
def profile_update(user)
  send_as :profile
  recipients user
  @battles=user.battles
  profile render(:partial=>"profile",
    :assigns=>{:battles=>@battles})
  profile_main render(:partial=>"profile_narrow",
    :assigns=>{:battles=>@battles[0..3]})
end
```

Now our users can choose to put our profile box wherever they think it looks best.

Profile Visibility

Earlier, we looked at using different `fb:if` tags to control visibility. Sometimes we may want to do this in the profile area. For instance, we may want to include a link that the profile owner can use to change the look and feel of their profile. Unfortunately, the `fb:if` tags can't be used in the profile area. Instead, Facebook provides several `fb:visible-to` tags that fulfill a similar purpose.



Figure 6.13: Users can add a profile tab for your application.

There are two big differences between the `fb:if` tags and the `fb:visible-to` tags. The `fb:if` tags control the existence of content. If the if condition isn't true, then no output is sent to the browser. The `fb:visible` tags control only the visibility of the content. When the conditional is false, the enclosed content is still written to the page; it is just hidden with CSS. If you view the source of the page, you can easily see content that is hidden to you. This means you shouldn't use the `fb:visible` tags to hide private information.

Additionally, all content on the profile is visible to the profile owner. That way, the profile owner will always know what their profile looks like. If you display different content to application users and nonapplication users, the profile owner will see both.

Profile Tabs

Along with displaying a small application box on their profiles, your users can choose to add an application tab to their profile. Application tabs show up at the top of the profile, as shown in Figure 6.13. An application tab is a cross between a normal canvas page and a profile box. When your application tab is clicked, a page you specify is loaded via Ajax and placed in the body of the profile, as shown in Figure 6.14, on the following page.

Pages displayed in a profile tab follow a special set of rules. First, tab pages aren't allowed to redirect. When a relative link is clicked on a page viewed in a tab, the linked page is requested via Ajax and loaded in the same frame as the first page.⁶ When a link is clicked with an absolute URL (a link with a hostname), the viewer is taken to the requested page.

When Facebook requests a page for a profile tab, two special parameters are sent. Facebook sends the `fb_sig_profile_user` parameter to tell our application which user the tab is being viewed for. Facebook also sends

6. There are many other rules specified at http://wiki.developers.facebook.com/index.php/New_Design_Tabbed_Profile.



Figure 6.14: Application tabs place your application on the profile.

the `fb_sig_in_profile_tab` parameter as a signal that a page view is for a tab. Like a normal canvas page, Facebook provides the ID of the user viewing the page if they have your application installed. If they don't, no user parameter is sent. Unlike a normal canvas page, the session identifier you are sent is a read-only session. You can use the provided session to retrieve information about the viewer, but you can't send notifications or perform other actions on their behalf.

For our users to be able to add a Karate Poke tab to their profile, we'll need to do a little configuration. Inside the Facebook Developer tool, we'll need to provide a URL used for profile tabs. Open the Developer application, and find the Profile Tab URL area. Let's use the path `/tab` for our tab page. Enter that as shown in Figure 6.15, on the next page.



Figure 6.15: Profile tabs are configured via the Developer application.

Once we've configured Facebook, we need to actually write some code. Let's start by creating a tab controller action. Since our tab view will be similar to our battles page, let's put it in the AttacksController. We saw earlier that the tab owner's ID will be passed in as the `fb_sig_profile_user`. Let's use that to load a list of battles for the user. Since we won't get the ID of the viewer, we'll also need to skip our filter that requires a user to be logged in:

Download chapter6/karate_poke/app/controllers/attacks_controller.rb

```
skip_before_filter :ensure_authenticated_to_facebook,
                  :only => [:index, :tab]

def tab
  @user = User.for(params[:fb_sig_profile_user])
  @battles = @user.battles
  render :action=>"tab", :layout=>"tab"
end
```

Now we just need to create a view for our tab page. Here is a simple view that should do the trick:

Download chapter6/karate_poke/app/views/attacks/tab.fbml.erb

```
<% if @battles.blank? %>
  <h1>Nobody has attacked <%=name @user%> yet.</h1>
  <p>
    Be the first.
    <%=link_to "Attack #{name @user} now!", new_attack_url%>
  </p>

<% else %>
  <%= will_paginate(@battles)%>
  <% for attack in @battles %>
  <div class="battle">
    <%= image_tag attack.move.image_name %>
    <%= link_to(
      name(attack.attacking_user, :linked=>false),
      battles_url(:user_id=>attack.attacking_user)) %>
    <%= attack_result(attack) %>
  </div>
</for>
</else %>
```

```

    <%= link_to(
      name(attack.defending_user, :linked=>false),
      battles_url(:user_id=>attack.defending_user)) %>
    with a <%= attack.move.name %>
  </div>
  <% end %>
<% end %>

```

Because our tab will be viewed inside the profile UI, we will want to change our layout a little. Our application's main navigation bar just adds clutter. Let's create a new layout in `app/views/layouts/tab.fbml.erb` to clean this up:

Download `chapter6/karate_poke/app/views/layouts/tab.fbml.erb`

```

<fb:fbml version="1.1">
  <%= stylesheet_link_tag "application"%>

  <%= yield%>
</fb:fbml>

```

There's just one more step to get this working. We need to set up a route for our tab view so that it appears at `/tab`:

Download `chapter6/karate_poke/config/routes.rb`

```
map.tab '/tab', :controller=>"attacks", :action=>"tab"
```

That should be it. Now you can go to your profile, click the plus sign to the right of your tab list (shown in Figure 6.13, on page 137), and add a tab for your application. If your application doesn't show up in the list, double-check your setup in the Developer application. You should now have a tab for your application. Click your application's tab to see your battle summary.

You should be able to click any of the links in the main battle list and be taken to the battles page of that user. You are taken outside the profile because the links in our tab view are absolute links created with the `battles_url` method.

If we had used `battles_path` instead, the page would have loaded in the profile. If you click any of the pagination links, you should see the new page loaded in the profile view.⁷

I mentioned earlier that a special read-only session is sent to us when an authenticated user visits a profile tab. We currently store our users' sessions in the database. If a user visits our application in a tab, their

7. You would if it weren't for a Facebook bug. See http://bugs.developers.facebook.com/show_bug.cgi?id=2646.

session will be overwritten by a read-only version. These read-only sessions will cause us problems in Section 9.4, *Move API Calls Out of Line*, on page 184. Let's change our `set_current_user` method to not save sessions from tabs:

```
Download chapter6/karate_poke/app/controllers/application.rb
def set_current_user
  set_facebook_session
  # if the session isn't secured, we don't have a good user id
  if facebook_session and
    facebook_session.secured? and
    !request_is_facebook_tab?
    self.current_user =
      User.for(facebook_session.user.to_i, facebook_session)
  end
end
```

With that done, we now have a complete implementation of profile tabs. This also wraps up our reworking of our users' profiles. We've added better content to our profile box and even allowed our users to add a profile tab to proudly show off their Karate Poke expertise. Next, we'll look at how we can test our code that uses the Facebook Publisher.

6.6 Testing Facebooker Publishers

Now that we've seen how to use the Facebook Publisher, let's look at how we can test our code. Ideally, we want to be able to run our publisher and look at the results without actually sending messages. After all, we can't run our tests very often if we get to send only twenty messages a day.

Back in Section 6.1, *Facebook Notifications*, on page 107, we saw a publisher method for creating a notification without sending it. We can use this feature in all of our tests. Let's start by creating a new file for our publisher tests. Since we're going to be testing the attack publisher, let's call this file `test/unit/attack_publisher_test.rb`.

Our file should include the normal Rails testing boilerplate:

```
require File.dirname(__FILE__) + '/../test_helper'

class AttackPublisherTest < ActiveSupport::TestCase
  fixtures :users, :belts, :attacks
end
```

Now that we have that, we can write our first publisher test. Let's start simple and test our attack notification code.

First, we'll need to ask our publisher to create a new notification:

```
def test_attack_notification
  story = AttackPublisher.create_attack_notification(attacks(:one))
  #assertions go here
end
```

Once we have our story, we need a way of making sure it matches our expectations. Since a notification has only one field, we can easily test that:

```
def test_attack_notification
  story = AttackPublisher.create_attack_notification(attacks(:one))
  assert_equal "<fb:fbml> ...", notification.fbml
end
```

Unfortunately, the returned notification object doesn't include the recipient list of the notification. To make sure our notification is sent to the right people, we'll have to use a mock:

```
def test_attack_notification
  a=attacks(:one)
  recipient=a.defender
  fm=flexmock(AttackPublisher)
  fm.new_instances.should_receive(:recipients).
    with(recipient.facebook_session.user)

  story = AttackPublisher.create_attack_notification(attacks(:one))
  assert_equal "<fb:fbml> ...", notification.fbml
end
```

That's all it takes to test a simple publisher. It's important to remember that these tests verify only that the published messages look like we expect them to look. They don't verify that the messages will work correctly when sent to Facebook. For instance, if you try to send a notification to a user who isn't friends with the sender, our tests will happily allow it while Facebook will raise an exception. Once you are satisfied that your publisher works the way you expect, you should verify the results through Facebook.

6.7 Summary

We've looked at quite a few features in this chapter. We started by sending out notifications when our users engage in battle. Next, we used the feeds to publicize our users' actions. After that, we implemented the profile publisher interface. We finished up by creating a place for talking trash and polishing our invitation system and profile area. Next, we'll look at how we can use JavaScript inside the Facebook canvas.

Chapter 7

Scripting with FBJS

Earlier, we looked at how to make our application more interactive by using the Facebook social features. Now, we're going to look at how we can use JavaScript to make our application more dynamic. If you've written JavaScript before, then Facebook JavaScript (FBJS) will look familiar to you. There are some important differences between normal JavaScript and FBJS. We'll look at these differences as we implement some new features in Karate Poke. We'll start by making our comment form a little more usable. We'll also look at creating Facebook dialog box messages. Finally, we'll use Ajax to allow users to post comments without a page refresh.

7.1 FBJS Overview

JavaScript is a relatively new feature for Facebook. At the time of the platform launch, there was no good way to make your application dynamic. Several months later, Facebook released FBJS.¹ You will very quickly notice some differences between normal JavaScript and FBJS. To begin with, Facebook renames all the methods and variables you create to effectively sandbox your code. By prepending all of your variables and methods with an application-specific prefix, Facebook makes it impossible for you to interact with another application's JavaScript. For example, if you create a method named `foo`, Facebook will rename your method to something like `α581937383_foo`. Additionally, Facebook changes the way your application interacts with DOM elements by

1. You can see basic documentation on FBJS at <http://wiki.developers.facebook.com/index.php/FBJS>.

removing direct access to certain attributes. This applies only to DOM objects, such as objects that represent `<div>` tags or `<p>` tags. Facebook does not change the way you interact with other built-in types, such as `String` and `Array` objects.

When developing FBJS, the open source Firefox plug-in Firebug² is an invaluable asset. If you haven't installed this plug-in for the Firefox browser, you definitely should. Firebug makes it easy to view your JavaScript as modified by Facebook. It also allows you to interactively modify DOM elements and debug your JavaScript.

A Simple Example

In Section 6.3, *Comments and Discussion Boards*, on page 124, we added a comment form to our battles page. Let's hide the comment form by default and make it appear when our user clicks a link.

If we were building this in a normal Rails application, our solution might look something like this:

```
<%= link_to_function "Talk some trash", "$( 'comment_form' ).show();" %>
<br />
<div id="comment_form" style="display:none">
  <% form_for Comment.new do %>
    <%= text_area_tag :body %> <br />
    <%= hidden_field_tag :comment_receiver,@user.id %>
    <%= submit_tag 'Post' %>
  <% end %>
</div>
```

Both the `$()` and `show()` methods come from the Prototype³ JavaScript library. Because of the implementation of JavaScript on Facebook, we can't use Prototype. Instead, we'll need to write raw JavaScript.

We can start by replacing the call to `$()` with `document.getElementById()`. Once we've done that, we need to find a way to make the `comment_form` element show up. Outside Facebook, we would write something like this:

```
document.getElementById( 'comment_form' ).style.display="block";
```

Facebook, however, removes our ability to access attributes of DOM elements directly. Instead, we'll need to use the setter functions it pro-

2. Available at <https://addons.mozilla.org/en-US/firefox/addon/1843>

3. You can find documentation on Prototype at <http://www.prototypejs.org/>. You can also check out the excellent screencast available at <http://peepcode.com/products/javascript-with-prototypejs>.

vides, `setStyle()`, in this case. That makes our finished code look like this:

```
<%= link_to_function "Talk some trash",
"document.getElementById('comment_form')"+
".setStyle('display','block');" %>
: <br />
<div id="comment_form" style="display:none">
  <% form_for Comment.new do %>
    <%= text_area_tag :body %> <br />
    <%= hidden_field_tag :comment_receiver,@user.id %>
    <%= submit_tag 'Post' %>
  <% end %>
</div>
```

That's a lot of additional code to write. Although this one line isn't too bad, you'll quickly be annoyed by all the JavaScript cluttering your views if you're forced to write it all by hand. Even though Prototype doesn't work with Facebook, Facebooker provides a JavaScript helper library that implements some of the most used Prototype methods. In fact, if we include the `facebooker.js` file in our layout, we can use the same JavaScript that we would use in a normal Rails application. Let's do that here:

```
<fb:fbml version="1.1">
  <%= render :partial=>"layouts/css" %>
  <%= javascript_include_tag "facebooker" %>
```

Once we've done that, our comment form becomes the following:

```
<%= link_to_function "Talk some trash",
"$('#comment_form').show();" %>
: <br />
<div id="comment_form" style="display:none">
  <% form_for Comment.new do %>
    <%= text_area_tag :body %> <br />
    <%= hidden_field_tag :comment_receiver,@user.id %>
    <%= submit_tag 'Post' %>
  <% end %>
</div>
```

After including `facebooker.js`, our code is back to what it would look like outside Facebook.

I mentioned earlier that Facebook filters our JavaScript. Let's take a look at what actually gets written to the page.

Our simple `link_to_function` turns into this:

```
<a href="#"
  onclick="(new Image()).src = &#039;/ajax/ct.php?app_id=5812356537\
&amp;action_type=3&amp;post_form_id=58892f4f71a44dc934baba03ca91\
a128&amp;position=3&amp;&#039; + Math.random();\
fbjs_sandbox.instances.a5812356537.bootstrap();
  return fbjs_dom.eventHandler.call(
[fbjs_dom.get_instance(this,5812356537),function(a5812356537_event)
{a5812356537_$(&#039;comment_form&#039;).show();; return false;},
5812356537],new fbjs_event(event));return true">Talk some trash</a>
<br />
```

That's quite a change from what we wrote. I don't even know exactly what most of it does. Out of that mess, our code is a relatively small portion:

```
a5812356537_$(&#039;comment_form&#039;).show();
```

During translation, Facebook replaces our call to `$()` with a call to `a5812356537_$()`. We'll see this prefix quite a bit. Facebook prepends all variable names and method names in our code with an `a` followed by our application ID. The good news is that this happens transparently to us. The bad news is that it makes our code harder to debug.

Living Without innerHTML

When we built our comment form, we used a string to store the comment body. Most databases have a relatively small limit to the amount of text that can be stored in a string. Let's add some JavaScript to count the number of characters entered into the form and warn our users when their comment is too long.

There are two main things we need to build this feature. The first is an event listener that gets called whenever text is typed into our comment box. We can add this like we would for any Rails application. We can modify our form to pass in listeners for both `change` and `keyup` events. We want to listen to both types of events to capture several cases. Some browsers call the `change` callback only when a field loses focus. Others don't call `keyup` when text is pasted. By using both callbacks, we make sure our count is accurate. Let's add a call to a function called `update_count`. We'll implement `update_count` shortly:

```
<%= text_area_tag :body, "",
:onChange=>"update_count(this.getValue(),'remaining');",
:keyup=>"update_count(this.getValue(),'remaining');"
%> <br />
<p id="remaining">&nbsp;&nbsp;&nbsp;</p>
```

Not Seeing Your JavaScript Changes?

As you're testing this, you may think your code isn't being updated. Facebook caches your JavaScript files by name. By default, Rails includes the time a file was last modified in its name. Unfortunately, it checks for updated timestamps only when the mongrel server is restarted. In development mode, you may need to restart your script/server each time you change your JavaScript files. For that reason, you may want to develop your JavaScript inside a script tag and move it to the application.js file only once it is complete.

If you see an error like "Failed to fetch required static file," that is Facebook telling you that you are trying to include too many JavaScript files. By default, Facebook allows you to reference only four external files. If you have more than that, you will need to combine them.

There's one important thing to note in this snippet. Instead of getting the value of a form element with `element.value`, Facebook requires you to call `element.getValue()`. If you're ever pulling your hair out trying to figure out why a value is undefined, this is one of the first places to look.

Now we need to implement our `update_count` method. We could define it inside a `<script>` tag in our page, but instead, let's add it to the `application.js` file.

There are just a few things our method needs to do. First, it needs to count the number of characters in our text field. Next, it needs to update a text area with a message.

```
function update_count(str,message_id) {
  len=str.length;
  if (len < 200) {
    // display the count with an okay message
  } else {
    // display the count with an error message
  }
}
```

Things get tricky at this point.

Along with removing the attributes of DOM elements, Facebook also removes the ability to set an element's `innerHTML`. To replace `innerHTML`,



Figure 7.1: `setTextValue` allows you to change a tag's content.

Facebook provides three different methods: `setInnerFBML`, `setInnerXHTML`, and `setTextValue`. We'll talk about `setInnerFBML` in a bit. For now, let's look at the other two.

Let's start by making our message a simple text string:

```
function update_count(str,message_id) {
  len=str.length;
  if (len < 200) {
    $(message_id).setTextValue(200-len+ " remaining");
  } else {
    $(message_id).setTextValue("Comment too long."+
      " Only 200 characters allowed.");
  }
}
```

If you run that code, you'll see an updated message underneath the form, like the one in Figure 7.1. Using only text is somewhat limiting. If we use the `setInnerXHTML` method, Facebook will allow us to replace the tag's content with sanitized XHTML. The string we pass to `setInnerXHTML` is sanitized according to the regular FBML rules. This means some tags are removed.

```
function update_count(str,message_id) {
  len=str.length;
  if (len < 200) {
    $(message_id).setInnerXHTML("<span style='color: green;'>"+
      (200-len)+" remaining</span>");
  } else {
    $(message_id).setInnerXHTML("<span style='color: red;'>"+
      "Comment too long. Only 200 characters allowed.</span>");
  }
}
```



Figure 7.2: `setInnerHTML` allows adding sanitized XHTML content to an element.

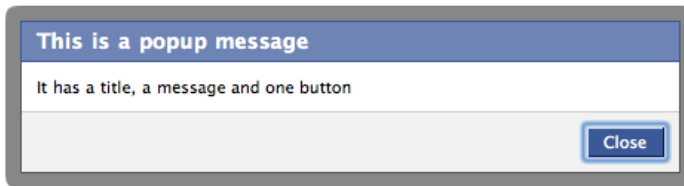
You can see the results of the previous code in Figure 7.2. When you use the `setInnerHTML` method, it's important to make sure you are using valid XHTML. If you try to use invalid XHTML, you will get a runtime error. You will also receive an error if you try to pass plain text to `setInnerHTML`.

FBJS in the Profile

So far, we've looked at using FBJS only in the canvas area. Facebook doesn't stop you from using JavaScript in your users' profiles, but this has some additional restrictions. To help keep the profile display clean and simple, Facebook restricts your JavaScript to being activated only after an element in the profile is clicked. This means you can't have JavaScript that runs on each profile display. Most uses of JavaScript already require a click, so this shouldn't cause any problems. It does mean you can't use an `onload` event to bypass Facebook's profile caching.

Displaying Dialog Boxes

Another frequent use of JavaScript in Facebook is displaying dialog box messages. Facebook provides two different types of dialog boxes. The first is a simple pop-up layer that is displayed in the center of the screen, as shown in Figure 7.3, on the following page. The other is a contextual dialog box that appears to come from a specific element on the page, as shown in Figure 7.4, on page 151.



[Click for popup](#) [Click for context](#)

Figure 7.3: Pop-up dialog boxes appear in the middle of the page.

Both styles of dialog box are created similarly by the Dialog class:

```
// use Dialog.DIALOG_POP for a popup
var d = new Dialog(Dialog.DIALOG_CONTEXTUAL );
// Setting the context is only
// necessary for contextual dialogs
d.setContext($('comment'));
d.onconfirm = function() {
    $('comment').setTextValue("");
};
// Show a message with only one button
d.showMessage(title,message,button_name);
// Or, show a message with two buttons
d.showChoice(title,message,confirm_name,cancel_name)
```

Once you've created the dialog box object, you can cause it to be displayed by calling either `showMessage` to show a message with one button or `showChoice` to show a message with two buttons. When a user clicks one of the buttons, the `onconfirm` or `oncancel` callback is executed.

Facebook dialog boxes are a really elegant way of showing error messages or requesting confirmation.

7.2 Ajax in FBJS

Let's turn our attention to one of the most popular uses of JavaScript: *Ajax*. Ajax originally stood for Asynchronous JavaScript and XML, a method for using JavaScript to update just a portion of a web page. We will look at how Ajax inside Facebook has evolved from its hum-



Figure 7.4: Contextual dialog boxes anchor to a specific element.

ble beginnings. We'll also look at the tricks necessary to get Rails and Facebook to work well together. To do this, let's make our comment form work asynchronously.

Mock Ajax

When the Facebook Platform launched, there was no way for your application to use JavaScript. Facebook provided several ways of implementing basic functionality, such as showing and hiding `<div>` tags,⁴ but no way of fetching content from your server. As a temporary workaround, Facebook introduced “Mock Ajax.” Mock Ajax uses attributes on elements to specify very basic Ajax functionality. Although Mock Ajax is no longer strictly necessary, its simple API can make it a nice tool for prototyping. I use Mock Ajax when possible because it often yields more readable code.

To convert our comment form to use Mock Ajax, we'll need to add three new attributes—`clickrewriteurl`, `clickrewriteform`, and `clickrewriteid`—to our submit button. When the element that has these attributes is clicked, Facebook will send the parameters found in the specified form to `clickrewriteurl`. The resulting FBML is then loaded into the element with the ID specified by `clickrewriteid`. When using Ajax via Facebook, we always want to use URLs that point directly to our server. This means using `comments_url(:canvas=>false)`.

4. You can add the `clicktohide` and `clicktoshow` attributes on an element to hide or show the specified IDs when that element is clicked. The documentation is at <http://wiki.developers.facebook.com/index.php/Clicktohide>. We don't look at those here because the same thing can be done more generally using JavaScript.

Make Your Application Intuitive and Interactive

Have you noticed how intuitive and interactive Facebook is? For example, when you click a user's photo in the search results screen, a window pops up that shows basic information about that user. Facebook gives you access to relevant information without making you navigate from screen to screen. Facebook has worked hard to make its site easy to use. Your application can benefit from this work.

By using tools such as Facebook dialog boxes and FBJS, you can make your application act and feel like Facebook. Instead of making your user navigate to a new page, consider showing more detailed information in a dialog box. Instead of requiring a page refresh to add a comment, make your comments section use Ajax. Not only will your application be easier to use, but it will feel more like Facebook as well.

First, we'll need to add these parameters to our submit tag:

```
<%= submit_tag 'Post',
:clickrewriteurl=>comments_url(:canvas=>false),
:clickrewriteid=>"all_comments",
:clickrewriteform=>"new_comment" %>
```

Next, we will want to add the all_comments <div> to our layout. The all_comments <div> will wrap all our comments. When a new comment is posted, we will rerender the whole collection. At the same time, we can move the existing comment display into a partial. This will prevent duplicating the comment display code.

```
<div id="all_comments">
  <%= render :partial=>"comments/comments" %>
</div>
```

With that done, we just need to make our controller handle Ajax requests. To Rails, Facebook Ajax requests look just like any other Ajax request. That means we can code this like we're used to doing:

```
def create
  comment_receiver = User.find(params[:comment_receiver])
  current_user.comment_on(comment_receiver,params[:body])
  if request.xhr?
    # pass true to the association to force a reload
    @comments=comment_receiver.comments(true)
    render :partial=>"comments"
```



```

else
  redirect_to battles_path(:user_id=>comment_receiver.id)
end
end
end

```

In the previous code, `request.xhr?` returns true if the request comes in via Ajax. Now, when a user clicks the submit button, the user's web browser will send an Ajax request to Facebook's servers. Facebook will then send the request to our server. Our server will send back just the comments area, and Facebook will update the page.

Using the Rails Ajax Helpers

Mock Ajax was a nice interim solution, but it isn't nearly as powerful as real Ajax. For one thing, there is no error handling. If your server fails to respond, the user won't know what happened. Additionally, you are limited to updating one element with FBML content. If you need error handling or want to use content other than FBML, Mock Ajax just isn't an option.

When Facebook added FBJS to the development platform, it included support for true Ajax requests through the Ajax object. Unfortunately, the implementation is somewhat limited. The Facebook Ajax implementation allows you to make calls to remote servers and return documents in FBML, JavaScript Object Notation (JSON), or a raw format. Notably absent is the ability to execute returned JavaScript. This means we can't run Rails `.rjs` templates. Even with these limitations, there is still quite a bit we can do.

Let's convert our comment form to use real Ajax. We'll start by removing the Mock Ajax attributes from the submit button. Next, we will replace our call to `form_for` with a call to `remote_form_for`. Since Ajax requests must go directly to our server, we'll need to change our call a little. We'll need to explicitly specify the URL to use in the form to include the `:canvas=>false` parameter:

```

<% remote_form_for Comment.new,
  :url=>comments_url(:canvas=>false),
  :update=>"all_comments" do |f|%>
  <%= text_area_tag :body, "",
:onChange=>"update_count(this.getValue(),'remaining');" ,
:keyup=>"update_count(this.getValue(),'remaining');"
  %> <br />
  <p id="remaining">&nbsp;&nbsp;&nbsp;</p>

  <%= hidden_field_tag :comment_receiver,@user.id %>
  <%= submit_tag 'Post'%>
<% end %>

```

That code should look familiar to you. It's the same code we would use for a normal Rails application. Facebooker takes care of hiding the Facebook-specific details from you.

We're now using real Ajax to do the same thing we did previously with Mock Ajax. Facebooker made the normal Rails helpers work for us, but only in a very simple case. To do much more using Ajax, we're going to have to look at the guts of the Facebook Ajax library.

Ajax Under the Covers

Let's take a look at how Ajax is implemented by Facebook. Facebook provides an `Ajax`⁵ object as an interface to the browser's `XMLHttpRequest` object. It provides a simple abstraction to your application, as you can see in the following code:

```
var ajax=new Ajax();
ajax.responseType=Ajax.JSON;
ajax.ondone = function(data) {
  // do something with the data
}
ajax.post('http://www.example.com/comments',
  {"body": "This is a comment","receiver_id": 4});
```

To make an Ajax request, we perform four steps:

1. Create an Ajax instance.
2. Choose a response type.
3. Provide a callback.
4. Call `ajax.post`.

We can create a new Ajax object using `var ajax=new Ajax();`. Next, we decide what type of data we will request. We can choose to receive the response as FBML, JSON, or raw content. If we choose FBML, Facebook will process the data into a format that can be passed to `setInnerFBML`. If we choose raw content, we could receive either plain text or XHTML to pass to the `setTextValue` or `setInnerXHTML` methods, respectively.

Once we've picked the type of data, we must give Facebook a method to call when the request is complete. We do this by setting the `ajax.ondone` method with a function reference. Usually, this method will do something with the data such as update the page. We can optionally provide a method to call when an error occurs by setting the `ajax.onerror` attribute. Finally, we call the `ajax.post` method to tell Facebook to start the request. There shouldn't be many times when you need to revert to

5. The documentation is at <http://wiki.developers.facebook.com/index.php/FBJS#Ajax>.

writing Ajax calls by hand, but understanding how they work can help you debug problems as they come up.

Using JSON with Ajax

So far, we've been limited to updating only one element at a time using Ajax. Let's look at how we can use JSON to update both the list of comments and also a message element with a single request. We'll start by changing our call to `remote_form_for` to make a request for JSON. To do this, we need to replace the `update` parameter with a success callback. Our success parameter is a JavaScript snippet that receives the JSON-formatted data in a variable named `request`:

```
<p id="form_message">&nbsp;&nbsp;&nbsp;</p>
<div id="comment_form" style="display:none">
  <% remote_form_for Comment.new,
    :url=>comments_url(:canvas=>false),
    :success=>"update_multiple(request)" do |f|%>
    <%= text_area_tag :body, "",
      :onchange=>"update_count(this.getValue(), 'remaining');" ,
      :onkeyup=>"update_count(this.getValue(), 'remaining');"
    %> <br />
```

With that done, we have two steps left to complete. We'll need to implement the `update_multiple` method and change our controller. Let's start with the controller method. We know that our `update_multiple` method will need to set the content of two DOM elements. We've used the `set_TextValue` and `setInnerHTML` methods in the past, but neither of those will work here. Our comment list includes several FBML tags that need to be expanded. We need some way of asking Facebook to convert a response from FBML to something we can pass to `setInnerHTMLFBML`. By convention, Facebook will treat any content in a variable whose name starts with `fbml` as an FBML document to be interpreted. Let's send back a JSON object with a couple of different fields. We'll include a list of IDs to be updated along with the content for each ID:

```
def create
  comment_receiver = User.find(params[:comment_receiver])
  current_user.comment_on(comment_receiver, params[:body])
  if request.xhr?
    @comments=comment_receiver.comments(true)
    render :json=>{:ids_to_update=>[:all_comments, :form_message],
      :fbml_all_comments=>render_to_string(:partial=>"comments"),
      :fbml_form_message=>"Your comment has been added."}
  else
    redirect_to battles_path(:user_id=>comment_receiver.id)
  end
end
```

Now we need to build the `update_multiple` method, which needs to loop through the list of IDs to be updated and then set the FBML for each element:

```
function update_multiple(json) {
  for( var i=0; i<json["ids_to_update"].length; i++ ) {
    id=json["ids_to_update"][i];
    $(id).setInnerFBML(json["fbml_"+id]);
  }
}
```

That's all there is to it. You could do quite a bit more using JSON. JSON is a great way of sending complex data to a JavaScript method on your page. For example, you could build a chat application that sends new messages as a JavaScript array.

Using fb:js-string

We've now looked at two different ways to use the `setInnerFBML` method. Not only can you pass it the results of an Ajax call of `Ajax.FBML` type or a string processed as FBML from a JSON request. You can also pass it an `<fb:js-string>`. `<fb:js-string>` is an FBML tag that creates a JavaScript variable. When Facebook processes the page, it turns the content of the tag into a JavaScript variable that can be passed to `setInnerFBML`.

These strings are used when you want to store some FBML that may be used later. For instance, if you show a number of thumbnails of images and want to show a larger image when the thumbnail is clicked, you could store the FBML for displaying the larger image in a `<js-string>`. Then, you can swap the content of the main display without having to go back to the server.

```
<fb:js-string var="photo_<%=photo.id%>_large">
  <%=image_tag photo.public_filename(:large)%>
  <%=photo.caption%>
</fb:js-string>
<%= image_tag photo.public_filename(:thumbnail),
:onclick=>"$('photos').setInnerFBML(photo_#{photo.id}_large);" %>
```

It is important to note that this tag creates an actual JavaScript variable. That means when you want to reference the result, you don't surround the name in quotes. For a photo with an ID of 7, this creates a variable named `photo_7_large`.

7.3 Summary

We've walked through a brief tour of FBJS. As you can see, it isn't as powerful as regular JavaScript, but it still can help you make your application more dynamic. FBJS is a relatively new feature, so expect it to continue to evolve over time.

Next, we'll look at how we can integrate our application with existing websites.

Chapter 8

Integrating Your App with Other Websites

So far, we've looked at making Karate Poke work only inside Facebook. Sometimes we'll want our application to be available outside Facebook. For instance, you may want to create a marketing page that people can view without having Facebook accounts. You also might want to take an existing application and make it available via Facebook. Finally, you may want to take advantage of features that aren't available through the canvas, such as image uploads or advanced JavaScript. We're in the homestretch now. This is the last new functionality we will add to Karate Poke!

We'll use Karate Poke to look at how to implement all this functionality. We'll start by creating KaratePoke.com, a site to promote our application. Next, we'll create a leaderboard that can be viewed both through Facebook and outside Facebook. We'll look at some of the special issues involved in sharing information. Next, we'll look at how to integrate Facebook with an existing application. We'll close by looking at how we can use the Facebook JavaScript library to get Facebook functionality outside the Facebook canvas.

8.1 Making Content Accessible

Let's start with the simplest case. Let's create a marketing site for Karate Poke. Our marketing page will explain what Karate Poke is and encourage users to join Facebook to play. We want this page to be available at <http://www.karatepoke.com>. Since this page is marketing for our application, we don't want it to appear inside the Facebook canvas.

Let's get started by creating a marketing controller. After we create the controller, we'll need some view files. You're welcome to create your own marketing page for Karate Poke. Alternatively, you can copy the versions I created from `chapter8/karate_poke/app/views/marketing` into your own views directory. With our views in place, we need to map the root URL to our marketing controller. That's done with the following entry in `routes.rb`:

```
map.connect '', :controller=>"marketing"
```

That's a problem. We're already using the default route for our battles page. We need a way to tell Rails to use one action for Facebook requests and another action for non-Facebook requests. Rails provides the `:conditions` option on routes to allow you to specify conditions that must be met for a route to be used. By default, you can make routes conditional only upon HTTP methods. Facebooker extends this functionality to include conditions about whether a request is from the Facebook canvas:¹

```
map.battles '',:controller=>"attacks",
           :conditions=>{:canvas=>true}
map.marketing_home '',:controller=>"marketing",
                  :conditions=>{:canvas=>false}
```

When a request comes in, Rails will look at whether the request is coming from Facebook or directly from a web browser. It does this by looking for the `fb_sig_in_canvas` and `fb_sig_ajax` parameters. If one of those exists, then the request is a Facebook request. Rails starts at the top of the `routes.rb` file and looks for a matching route. Since Rails matches from the top down, you should always make sure your most specific route is first. Let's consider the following route:

```
map.battles '',:controller=>"attacks"
map.marketing_home '',:controller=>"marketing",
                  :conditions=>{:canvas=>false}
```

The first route will match all requests for the default route, so no requests will ever be sent to our marketing controller. If instead we were to reverse the order, non-Facebook requests would go to our marketing page, and Facebook requests would be sent to the battles page:

[Download](#) `chapter8/karate_poke/config/routes.rb`

```
map.battles '',:controller=>"attacks",
           :conditions=>{:canvas=>true}
map.marketing_home '',:controller=>"marketing"
```

1. You can read about how this works in an article by Jamis Buck at <http://weblog.jamisbuck.org/2006/10/26/monkey-patching-rails-extending-routes-2>.

With those routes, we almost have a functioning marketing page. Due to our `ensure_authenticated...` filter, our application will redirect viewers to the Facebook application install page. To make our marketing page visible outside Facebook, we will need to skip that filter using `skip_before_filter :ensure_authenticated_to_facebook`. If only a small section of your application is used inside Facebook, you probably want to move the call to `ensure_authenticated_to_facebook` from the ApplicationController to the controllers that handle Facebook requests.

Using conditional routing works nicely when we want two pages with the same URL to have different functionality. Next, we'll look at using the same logic with different displays.

8.2 Actions That Work Both Ways

Sometimes we want to do more than have different actions for the same URL. Sometimes we want to have the same logic and just have different displays. We're going to add a little more content to our Karate Poke marketing site. We will build a leaderboard, a list of the top users ordered by the number of successful battles they've engaged in. We'll look at how we can use one controller action with different displays for Facebook and non-Facebook requests.

By now, you could probably build the Facebook version in your sleep. We can create a LeadersController and add a route to `routes.rb`:

```
map.resources :leaders
```

With our routing in place, we just need an action and a view:

```
Download chapter8/karate_poke/app/controllers/leaders_controller.rb
```

```
def index
  @leaders = User.paginate(:order=>"total_hits desc",
                          :page=>(params[:page] || 1))
end
```

```
Download chapter8/karate_poke/app/views/leaders/index.fbml.erb
```

```
<%= will_paginate @leaders%>
<ul>
< for leader in @leaders %>
<li><%=name leader%>: <%=leader.total_hits%></li>
< end %>
</ul>
```

That's nothing new to us. Now we just need to figure out how to make that available to non-Facebook users. You've probably noticed that we have been creating `.fbml.erb` templates for all our views. Rails will try to

send back the right content type for each request. That means supporting both Facebook and non-Facebook requests in a single action can be as simple as creating another template with a different file extension. If we build an `.html.erb` template, Rails will use that for non-Facebook requests. Here is a version of our leaderboard that we can use outside Facebook:

```
<%= will_paginate @leaders%>
<ul>
<% for leader in @leaders %>
<li>Unknown: <%=leader.total_hits%></li>
<% end %>
</ul>
```

To view the HTML version of our leaders page, go to `/leaders` using your callback URL as the host. When you do, you'll be redirected to the Facebook application install page. We forgot to tell Facebooker that our users don't need to be logged in to view this page. If we add `skip_before_filter :ensure_authenticated_to_facebook` to the beginning of our controller, we should be able to view our page. This is something we'll need to do on every page that is visible outside Facebook.

As you can see, this looks similar to the Facebook version of the page. Since web browsers don't understand FBML, we had to remove our code that rendered an `<fb:name>` tag. In fact, we don't have any way of displaying our users' names. Facebook won't let us store their names in our database, and we can't make an API request to retrieve them, since non-Facebook requests don't have a Facebook session.

This is a tricky problem. It's nice that Facebook gives us access to a wealth of information about our users, but it locks us in to Facebook. Different applications will solve this problem in different ways. If your application already exists outside Facebook, you may be able to use your existing data. In our case, we really need a name to show for each user. We'll look at fixing that next.

8.3 Handling Facebook-Specific Data

I wish there was some magic bullet I could give you to make handling Facebook-specific data easy. Unfortunately, I can't. The cost of getting access to the wealth of Facebook data is that it can be used only inside the context of Facebook. In our case, we don't need much information from our users outside the canvas; all we need is a name for each user.



Figure 8.1: Asking for data to be shown outside Facebook

Although Facebook limits what API data we can store, it doesn't limit what information we can get from our users. If we want to display a name outside Facebook, we can simply ask for it. We can even have a little fun with this. We can ask our users to give us a nickname that they want displayed. We can use this nickname both outside Facebook and as a replacement for "a hidden ninja."

We'll start by adding a nickname field to our users table. After we've created and run that migration, we'll need to build a form. Since we'll need to gather this data from all our existing users, we should put our form front and center. Let's add it to our battles page. Once a user has set their nickname, we can hide the form and use the Ajax we learned in the previous chapter to show it on demand. You can see what we're going to build in Figure 8.1.

Let's start by creating a simple controller method. Since we are going to be updating a user object, let's create a users controller. Make sure you set it up as a resource in routes.rb. We will use the update action to perform the update.

```
Download chapter8/karate_poke/app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController
  def update
    saved = current_user.update_attribute(:nickname, params[:nickname])
    # the update was a success, show the closed_form
    render :partial=>"nickname", :locals=>{:closed=>saved}
  end
end
```

Next, we'll need a view. Our view will serve two purposes. We'll want it to show the nickname form when a user hasn't set their nickname. We'll also use it to display their nickname once they have one. By passing

in a local variable to our view, we can control whether to show the nickname form, as you can see here:

```
Download chapter8/karate_poke/app/views/users/_nickname.erb
<div id="nickname_open"
  <% if closed %> style="display:none" <% end %> >
<% remote_form_for :user,current_user,
  :url=>user_url(:id=>current_user,:canvas=>false),
  :html=>{:method=>:put},
  :update=>"nickname" do |f| %>
  Select a nickname to show outside of Facebook
  <%= text_field_tag :nickname,current_user.nickname%>
  <%= submit_tag "save"%>
<% end %>
</div>
<div id="nickname_closed"
  <% unless closed %> style="display:none" <% end %>>
  Your nickname is <%=current_user.nickname%>
  <%=link_to_function "(change it)",
    "$('nickname_open').show();$('nickname_closed').hide()"%>
</div>
```

Now we just need to add that to our battles view:

```
<% fb_if_is_user @user do %>
  <div id="nickname">
    <%= render :partial=>"users/nickname",
      :locals => { :closed => !current_user.nickname.blank? }%>
  </div>
<% fb_else do %>
  ...
```

Next, we can make our name helper use the nickname field. We also need to create a helper for displaying names outside Facebook:

```
Download chapter8/karate_poke/app/helpers/application_helper.rb
def name(user,options={})
  fb_name(user,
    {:ifcantsee=>(user.nickname||"a hidden ninja")}.merge(options))
end

def external_name(user)
  user.nickname || "a hidden ninja"
end
```

There's just one final step in this process. We need to change our leaderboard to use the `external_name` helper when we are outside Facebook:

```
<%= will_paginate @leaders%>
<ul>
<% for leader in @leaders %>
```

```
<li><%= external_name leader %>: <%=leader.total_hits%></li>
<% end %>
</ul>
```

That was quite a bit of work for just one piece of data. This works when we need information about a user, but it would be nearly impossible to do the same thing with an event. After all, events can change outside our application, and there would be no way to keep our information up-to-date. As you can see, the cost of access to Facebook's data is being locked in to using it through its site.

8.4 Sharing Sessions

We've seen how to make our application available to both Facebook and non-Facebook requests. At times we may need to move between Facebook requests and non-Facebook requests. For instance, since Facebook doesn't allow multipart form posts, we'll need to bypass Facebook for image uploads. We also may want to bypass Facebook to use more advanced JavaScript functionality, such as drag and drop.

In most cases, we can solve this problem using the `<fb:iframe>` tag. An *iframe*, or inline frame, is an HTML component that embeds content from a URL into an element within another page. The content included in the iframe is loaded by the browser from the remote server and can include any content that an HTML page can contain. By including an iframe in your canvas page, you can embed regular HTML content including Flash, JavaScript, or even Java.

The `<fb:iframe>` FBML tag creates an iframe inside our canvas. It takes a single parameter, `url`, which is the URL on our server that will provide the content for the frame. Make sure you specify a URL with `:canvas => false` so that the request goes directly to your server. Facebook modifies the URL you provide to `<fb:iframe>` to include all the normal Facebook parameters, including `fb_sig_session_key`. This means the requested page will have access to the `facebook_session` of the viewer. In fact, because of the way Facebooker implements sessions, once a session has been established with an `<fb:iframe>`, all direct requests from that user's browser will maintain that session.

We will have to make one change to make session sharing work. When we originally configured our application, we used the Rails cookie session store. Because our application's canvas and iframe pages have different URLs, our users' browsers won't allow us to use the same cookie for both. To work around this, we can use a different type of session.

We'll use the ActiveRecord store to keep our session information in the database.

First, we'll need to run `script/generate session_migration` to create the sessions table. You should also run `rake db:migrate` to execute the migration. Next, we'll need to uncomment a line in `environment.rb`:

```
config.action_controller.session_store = :active_record_store
```

With that done, our session can be shared between the Facebook canvas and an iframe. We don't need to use an iframe to share session information, though. We can also join the session by including session information in a link directly to our server. For example, if we wanted to link from our FBML leaderboard to our HTML leaderboard without sharing sessions, we could use this:

```
<%= link_to "HTML view", leaders_url(:canvas=>false)%>
```

If we wanted to do the same thing while maintaining the user's session, we would use this:

```
<%= link_to "HTML view",  
  leaders_url(facebook_session_parameters.merge(:canvas=>false))%>
```

By adding this parameter, we're telling Facebooker which existing session to use. Once the link between the Facebook session and the direct session has been established, the link will continue to exist even without sending `facebook_session_parameters`.

If we had an existing application with a login system, we could use an iframe to link a user's Facebook account to their existing information. By having our user log in to our application within an `<fb:iframe>`, we would have access to both their Facebook session and their existing account information. Once we have their Facebook ID saved, we could share information between a Facebook version of our application and a non-Facebook version.

8.5 Accessing Facebook Outside the Canvas

We've looked at several ways of integrating Facebook and non-Facebook applications. Everything we've looked at so far has focused on the server side. If our application already exists entirely outside Facebook and we want only basic integration, we have another option to consider.

Facebook has recently released a client-side JavaScript library that provides access to the Facebook API from any website. We could use the

Sharing Information with Facebook

We've spent a lot of time looking at all the information that Facebook can offer us. Don't limit yourself to just this information. If you have an existing application, make use of your existing data. For instance, the Ultimatums application allows users to share data between their Facebook application and its existing website.

Similarly, the Twitter application allows users to update their Facebook status by sending a message to Twitter. Communication doesn't need to be a one-way street. Don't just integrate your application with Facebook; use the platform to integrate Facebook with your existing application.

JavaScript API to give us access to our users' social graphs or even to send notifications from the browser.

The Facebook JavaScript API requires some configuration before it can be used. For security purposes, web browsers limit JavaScript to making requests only to the originating server.

Since the Facebook API needs to communicate with Facebook, we'll need to help it work around the browser security model. To do this, we'll create a file called `xd_receiver.html` in our public directory. The file should have the following content:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
  <head>
    <title>Cross-Domain Receiver Page</title>
  </head>
  <body>
    <script \
src="http://static.ak.facebook.com/js/api_lib/v0.3/XdCommReceiver.debug.js" \
    type="text/javascript"></script>
  </body>
</html>
```

Now that we have that page in place, we can create a page that uses the Facebook API library. Let's use our marketing page to experiment with the library. We'll start by including it into our page.

Welcome to Karate Poke

```
Get first name of user : {Array}
  [0]: {Object}
      first_name: Alberto
      uid: 563710619
```

Figure 8.2: Facebook JS API showing a user

That takes only one line of code:

```
<script
src="http://static.ak.facebook.com/js/api_lib/v0.3/FeatureLoader.js" \
  type="text/javascript"></script>
```

To use the library, we'll need to create an instance of the `FB.ApiClient`. Its constructor takes two parameters, our application's API key and the server path to our `xd_receiver.html` file. Before we can make API calls, we'll need to make sure our user is authenticated to Facebook. To do this, we'll call the `requireLogin` method. If this is the first time we're using the API for a user, they will be redirected to a Facebook login screen where they are prompted to enter their login information. Once they have authenticated with Facebook, they will be sent back to our site where execution will continue.

Since the JavaScript API requires our user to be authenticated with Facebook, the `requireLogin` method requires that a callback function be provided. This callback will be called once authentication is complete. You'll see extensive use of callback functions inside the API client. In general, we want all our code that depends upon the Facebook API to be called from this callback. The following example opens an alert dialog box containing the Facebook ID of the logged-in user; give it a try:

```
// Create an ApiClient object, passing app's API key and
// a site relative URL to xd_receiver.htm
var api = new FB.ApiClient('<%=Facebooker.api_key%>',
                          '/xd_receiver.html',
                          null);
api.requireLogin(function(exception) {

    alert("Current user id is " + api.get_session().uid);
});
```

I've had good luck getting that code to work in Firefox, but not in Safari for the Mac. The problem seems to occur only when working using `script/server`. I haven't seen the same issues once the application is deployed. You'll want to keep this in mind while you are doing your development.

We've seen how to get the Facebook ID of the current user. Now let's take a look at how to get the user's name. We'll use the `users.getInfo` API call.² Our code will need to make a call to this API function and pass in three parameters: the Facebook ID of the person whose name we want, the fields we want to retrieve, and a callback to be called when the request completes. The following is an example. If you run that code, the result should look like Figure 8.2, on the previous page.

```
api.requireLogin(function(exception) {  
  
  // Get the name of the current user  
  api.users_getInfo(api.get_session().uid,  
    ["first_name"],  
    function(result, exception) {  
      Debug.dump(result, 'Get first name of user ');  
    });  
});
```

In the previous code, we display the result of our API call using the `Debug.dump` method. This method writes data to the browser's JavaScript console. If you would rather see the result on your web page, you can include a text area with the ID `_traceTextBox` like in the following code:

```
<textarea style="width: 600px; height: 300px;" id="_traceTextBox">  
</textarea>
```

This code is very different from our Ruby code that does the same thing, `user.first_name`. Along with having to deal with callback functions, we also have to deal with an awkward naming convention. You'll find that these method names very closely match the ones used in the official Facebook PHP client.

We're not limited to such basic data retrieval. We can make much more complicated calls. For instance, we could get the names of all a user's friends by nesting an API call inside our callback.

2. You can find documentation at <http://wiki.developers.facebook.com/index.php/Users.getInfo>.

This is shown here:

```
// require user to login
api.requireLogin(function(exception) {

  // Get the names of friends
  api.friends_get(null, function(result, exception) {
    api.users_getInfo(result, ["first_name"],
      function(result, exception) {
        Debug.dump(result, 'Get first name of friends ');
      });
  });
});
```

That returns an array containing the name of each friend of the user. We can even send notifications using this API.³ For the most part, everything you can do with the server-side API can be done via JavaScript.

8.6 Summary

We have now looked at the major issues involved in integrating your existing website with Facebook. We have seen how to configure your Rails application to send requests for the same URL to different actions depending upon whether the request comes from Facebook, and we have seen how to write an action to handle both Facebook and non-Facebook requests. We looked at using iframes for integration as well as how to share sessions. We ended with a brief tour of the Facebook JavaScript API.

Congratulations! Together we built a complete Facebook application. We've covered quite a bit of ground, but we're not done yet. Next, we're going to turn our attention to performance. It's time to see how to make our application handle the heavy load we hope to receive.

3. You can download the documentation for the JavaScript API from http://wiki.developers.facebook.com/images/6/64/Facebook_JavaScript_API_Documentation.zip.

Chapter 9

Scaling and Performance

Over the past eight chapters, we've built a complete Facebook application. We've seen how to make our application enticing to users. If all goes well, it will attract a large number of people. Making our application handle millions of users can be a real challenge. Even if you don't have millions of users, you still need to think about your application's performance. For example, you still have Facebook's eight-second timeout to worry about. You may also be able to host your application on less hardware, saving you money.

For the most part, tuning a Facebook application is like tuning any other Rails application. We'll look at some of the most important performance issues here. We'll start by looking at ways to eliminate database queries. From there, we will see three different styles of view caching. We'll finish up by looking at some ways to reduce the performance penalty from Facebook API calls.

9.1 Getting Faster with Memcached

In Section 3.8, *Refactoring and Performance*, on page 70, we looked at several methods of speeding up our application. In particular, we added indexes to make our queries faster and used `include` to reduce the number of queries that were run. Although those two changes can reduce the number of database queries our application needs, they can't eliminate the queries altogether. In fact, we know that we will be running at least one query on every page view, because our `set_current_user` method needs to look up the current viewer. Since our `User` object likely doesn't change between requests, that seems like a waste of resources. We can eliminate repeated database queries for the same data by using *memcached*.

Memcached is an in-memory caching system created by the developers of LiveJournal.com.¹ You can think of memcached as a giant hash in the sky. It allows you to get and set values from a cache that is shared between all your web servers. Memcached stores all its data in memory, which makes it blazingly fast. It also means that if the server it is running on crashes, all the stored data will be lost. Consequently, we can't use memcached in place of a database. We can, however, use it to cache data retrieved from a database.

Memcached can give a big performance improvement, but it doesn't come inexpensively. Memcached will require at least some dedicated hardware. Because memcached stores data in memory, it requires a server with at least several gigabytes of RAM. If you are using a shared hosting environment or a virtual private server, memcached is not for you.

You have to install memcached on your development machine to be able to use it during development. It's available for most platforms, including Windows.² Once you've installed memcached, you'll need to install the Ruby client by running `gem install memcache-client`. Along with installing the client, we'll also want to install the `cache_fu` plug-in, available at `svn://errtheblog.com/svn/plugins/cache_fu`. After installing the plug-in, we can make our User model cacheable by calling `acts_as_cached`. We use the `:find_by` option to tell `acts_as_cached` that we want to use the `facebook_id` for finding our objects.

```
class User < ActiveRecord::Base
  acts_as_cached :find_by=>:facebook_id
  ...
end
```

Once we've made our User cacheable, we'll get access to some new methods. We'll use one of these, `get_cache`, to rework our `find` method. The `get_cache` method takes a single parameter, the cache key to use to look for cached objects. It also takes a block. If the object can't be found in the cache, `cache_fu` will execute the block and store the returned object in the cache.

```
def self.for(facebook_id, facebook_session=nil)
  u=get_cache(facebook_id) do
    find_by_facebook_id(facebook_id)||
    create_by_facebook_id(facebook_id)
  end
end
```

1. More information is available at <http://www.danga.com/memcached/>.

2. The Windows version is available at <http://jehiah.cz/projects/memcached-win32/>.

```

    returning(u) do |user|
      unless facebook_session.nil?
        user.store_session(facebook_session.session_key)
      end
    end
  end
end

```

With that code, our `for` method will look for a user in memcached. If one can't be found, it will then look for that user in the database. If it still can't be found, the user object will be created. Our code will still hit the database the first time a user is loaded. After that, the User object will be pulled from memcached as long as it remains in the cache. Because memcached stores everything in memory, it will eventually need to remove old objects when it runs out of room for new objects.

Now that we have users being stored in the cache, we'll need a way to remove them when they change. We'll do this with an `after_save` callback. Our callback will call `expire_cache` to remove an object from memcached, as shown in the following code. We want to make sure our filter method returns `true`. If it were to return `false`, ActiveRecord would stop running any other callbacks that may be defined.

```
after_save :expire_fb_cache
```

```

def expire_fb_cache
  returning true do
    expire_cache(facebook_id)
  end
end

```

Unlike ActiveRecord, memcached can store more than just a single record at a time. In Karate Poke, we'll often need to know a user's belt and their available moves. If we were to load those in our `for` method, they would be cached along with the user object. This would eliminate two more database queries.

```

def self.for(facebook_id, facebook_session=nil)
  u=get_cache(facebook_id) do
    find_by_facebook_id(facebook_id,
      :include=>[:user, :available_moves])||
    create_by_facebook_id(facebook_id)
  end
  returning(u) do |user|
    unless facebook_session.nil?
      user.store_session(facebook_session.session_key)
    end
  end
end
end

```

Using memcached can be a great way to eliminate database queries, but it isn't a magic bullet. It takes time for memcached to store objects in the cache. Unless your application repeatedly pulls the same data from the cache without modifying it, you may even experience a slowdown. For example, if you store a last-accessed date on each user and update it on every request, you won't get any benefit from caching.

9.2 Caching Our Views

Caching our models can reduce database queries, but there is more we can do. We'll look at several methods of view caching that can eliminate dynamic requests altogether. Rails provides three different levels of view caching, each with its own pros and cons. Although page caching gives you the biggest performance benefit, it is also the most difficult to manage. Fragment caching is easy to use, but it does relatively little to boost performance in the average case. Of the three, action caching typically gives the largest bang for your buck. If we combine view caching with memcached, we get the best of both worlds. We will eliminate code when we can, and when we can't, we will make it faster with memcached. Thanks to the power of the Facebook platform, we can use FBML to customize the look of a cached page as it is displayed to the user.

Page Caching

Page caching is the sledgehammer of Rails caching; it is incredibly powerful and imprecise. With page caching enabled, Rails will write the results of each request into the public directory. If your web server is correctly configured,³ future requests for this page will be served from disk, completely bypassing Rails.

Page caching is by far the fastest style of caching. Page cached pages are served directly from the web server, bypassing Rails completely. A typical web server can comfortably serve 1,000 static files per second. As a side effect, there is no way to ensure that the viewer has permission to access the requested page. Additionally, because the same content will be sent to every user, a normal Rails application has no way of customizing the view. For instance, with a page-cached page, you can't include a customized greeting in the header. This limits the number of places where page caching can be used.

3. You can see an example configuration in the fantastic Rails caching tutorial at <http://www.railsenvy.com/2007/2/28/rails-caching-tutorial#apache>.

Thanks to the power of FBML, we can page cache many more pages than a typical Rails application. In fact, we can page cache just about every page where each user sees the same basic content. In my previous example, we could easily include a welcome message for each user by using the `<fb:name>` tag. If you specify `loggedinuser` for the `uid` of an `<fb:name>` tag, Facebook will display the name of the current viewer.

In most Rails applications, you can't use page caching if you need to verify that a user has access to a requested page. That isn't the case for our Facebook application. We can verify that a user has installed our application by wrapping an `<fb:redirect>` tag in the `else` condition of the `<fb:if-user-has-added-app>` tag. This allows us to verify that all viewers are logged in while still serving the page straight from disk.

Configuring page caching is incredibly easy. To cache the `index` action of our `marketing` controller, we include the following code:

```
cache_page :index
```

When the `index` action is run, the content will be stored in the file `public/marketing/index.html`. When we want to remove that page from the cache, we just call `expire_page :index`.

Although page caching is very simple, you need to keep several issues in mind. Page caching works based upon the URL of a request. When a cache file is written, all query parameters are discarded. Two requests for the same URL with different query parameters will use the same cached file.

Additionally, because cached pages are stored on disk, page caching can be tricky in a multiserver environment. When a page is expired from the cache, the cache file will need to be removed from every server. This means you'll need a shared filesystem for the `public` directory. Page caching quickly becomes difficult to manage as your application spreads to multiple servers.

In *Karate Poke*, several pages could easily be page-cached. Our marketing pages, for example, are basically static pages. Our leaderboard page is another good candidate for page caching, because it requires a database query and can be updated only a couple of times per day. Other pages, like our new attack form, aren't a good match for page caching because each user sees a different set of available moves.

Action Caching

Action caching is similar to page caching. At the end of an action, Rails writes a full copy of the content to a file in the public directory. Unlike page caching, Rails still processes action-cached requests. When a request for an action-cached page comes in, Rails will run all filters associated with the action. If none of the filters render or redirect, Rails then serves the previously stored page content.

Because action-cached requests still go through Rails, they are not nearly as fast as page-cached pages. Still, there are several benefits over page caching. First, Rails filters can be run to make sure the viewer has permissions to access the requested page. Second, because we are checking for cached content inside Ruby code, we can store our cached pages somewhere other than on disk. In a clustered environment, action-cached pages are often stored in memcached.⁴

Configuring action caching is very similar to page caching. To cache the same index action, we would use this:

```
cache_action :index
```

Similarly, cached actions are expired by calling `expire_action :index`.

To store cached actions in memcached, you can set the `fragment_cache_store` in your `production.rb` file. Even though the parameter is `fragment_cache_store`, your setting will be used for both fragment and action caching.

```
ActionController::Base.fragment_cache_store =  
  :mem_cache_store, "memcached_server:11211"
```

As you can see, using action caching is similar to page caching. It works in similar situations while giving you the flexibility to run code in filters. It also scales more easily in a multiserver environment. In Karate Poke we would consider action caching the same pages we considered page caching.

Fragment Caching

We have looked at two methods for caching that bypassed our action's code altogether. The third style of view caching, fragment caching, works differently. Instead of bypassing the action, fragment caching

4. Instructions on storing the session in memcached are available at <http://wiki.rubyonrails.org/rails/pages/HowtoChangeSessionStore>. You'll also want to look at <http://www.elevatedrails.com/articles/2008/07/25/memcached-sessions-and-facebook/>.

Be Prepared to Scale

The rapid growth of many popular Facebook applications is both a curse and a blessing. Because of viral growth and the power of the social network, an application will occasionally become popular almost overnight. For example, the Friends for Sale application grew from 1 million page views a day to 10 million page views a day in about two months. (That's 200 requests per second!)

Although not every application catches on, those that do catch on tend to grow quickly. You don't need to spend a lot of time making your application scale to millions of users, but it helps to understand the basic techniques that can help you scale. While you're building your application, think about how it will perform. If there are easy changes you can make to allow it to scale better, make them!

is used to bypass a portion of the view code. Fragment caching is normally used to bypass expensive view calculations.

To cache a fragment of a view, we wrap our code in a cache block. The following code will cache the creation of our leaderboard:

```
<% cache :leaders do %>  
  render :partial=>"leaderboard"  
<% end %>
```

If there is content already stored under the name `leaders`, the code in the block will not be executed, and the cached content will be used instead. If no content is found, the block will be executed, and the resulting fragment will be stored in the cache. You remove a fragment from the cache with a call to `expire_fragment`.

Fragment caching is easy to use but also provides the smallest benefit. Because fragment caching happens in the view, your application will still spend time loading data in the controller.

In Karate Poke, we might consider using fragment caching on our user's battle page. We want to render the new attack form separately for each user, since they have access to different moves. The battle list is displayed the same way for everyone and could be easily fragment cached.

9.3 Caching with refs

We've seen how to cache our objects using memcached and also how to cache our views using the built-in Rails caching. In addition to these, Facebook gives us the `<fb:ref>` tag for caching. Facebook refs provide a method for setting content for a key and then displaying that content in an FBML page. In many ways, Facebook refs are like a version of the Rails fragment cache that stores data on Facebook's servers.

There are two typical uses for refs. The first is view caching. We can use refs in a manner similar to the way we used fragment caching earlier. We can store expensive views in a ref to avoid rendering them for each request. We can also use refs to allow us to update multiple pages at once.

The Mechanics of refs

Facebook provides two different types of refs, URL refs and handle refs. Both provide the same functionality and differ only in how they get their content.

URL-based refs use an HTTP URL as their access key and store data by fetching it from your server. This sounds promising, but they have several issues. To create a URL-based ref, your application calls `facebook_session.server_cache.refresh_ref_url` and passes in a URL. Facebook will then make an HTTP request to the supplied URL and will store the response. To display the content of the ref, you use the `<fb:ref>` FBML tag, supplying the same URL.

Although it isn't documented, it appears that URL refs are limited to containing just 4KB of data. Attempting to store more data than this will result in your content being silently discarded. Additionally, it appears that URL-based ref updates time out very quickly. Instead of the normal eight seconds, URL ref updates appear to time out in less than a second. If the update fails, you receive no notification, and no content will be stored. Finally, URL refs are difficult to test in development mode. When you run `script/server`, Rails starts only a single process. If you try to update a URL ref during an HTTP request, you will end up with a deadlock. Your application will contact Facebook, which will make a request to your server. Because your server is already executing a request, the request from Facebook will hang.

Because of these issues, I don't recommend using URL refs. Instead, use handle refs. Handle refs are set with the `facebook_session.server_`

cache.set_ref_handle method. This method takes two parameters, a handle and the content to store. Because you specify the content to be stored at the time of the call, you can set handle refs on a development server. There is still a limit to the amount of data that can be stored in a handle ref, but it appears to be much larger than for URL refs. Unfortunately, neither of these limits is documented by Facebook. They have been observed empirically, however.

Once you've stored content for a ref, you can use the `<fb:ref>` tag to include that content in an FBML page. Refs can be displayed both in the profile area and in the canvas area. There is no limitation to the content that can be stored in a ref; they can even contain other refs.

Typical Uses for refs

Although we can cache view data in refs, certain limitations make it more difficult than using fragment caching. Rails fragment caching can detect whether cached content already exists and replace that content on request. Because Facebook refs are write-only, there is no way to see whether content for a given ref exists. That means we'll need a way to ensure that our cached content is sent to Facebook at appropriate times. Unlike Rails caching, the only way to clear an old cached value is to provide a new one.

Along with using refs to avoid the cost of rendering a view, you can also use refs to update multiple pages at once. For instance, if you were building a news application that showed a list of stories on your main page, you would probably want to cache that page. If you used Rails action caching, you would need to clear the cache each time a story changed. That's not too bad. If you also wanted to show the number of comments on each story, you would need to clear the cache each time a comment was left on any front-page story. Suddenly, you've lost the benefit of caching.

Instead, you could store the number of comments in a ref. Our view could look something like this:

```
<% for story in @stories %>
  <%= display_title(story) %>
  <%= display_summary(story) %>
  <fb:ref handle="comment_count_<%=story.id%>" />
<% end %>
```

Now, when a new comment is made for the story with an ID of 10, you simply change the value stored in the ref called `comment_count_10`. You

can action cache your main page and still have up-to-date comment counts in real time. You can use refs in a similar manner for displaying the scores of anything that is voted on, movie ratings, or any time a dynamic attribute is mixed with mostly static content.

This style of caching becomes an even greater win when the data in question is displayed on your users' profiles. If you display a user's favorite movies on their profile and include the average rating of that movie, you could conceivably need to update a very large number of profiles each time that movie receives a new score. If instead you were to store the movie's average rating in a ref, you could update every profile at once.

Facebook refs don't provide any magic bullet to make your application perform better, but they are a powerful tool to have in your toolkit. They are a nice complement to the Rails built-in caching helpers.

9.4 API Performance

Now that we've used memcached and view caching to speed up our application, there is only one major slowdown we need to eliminate. When our code makes an API call, our server is sitting idle waiting for a response from Facebook. If we eliminate this dead time, our server will be able to handle more requests with the same amount of resources.

We'll start by looking at the Facebook Query Language (FBQL) as a way to improve data retrieval performance. Next, we'll look at an alternative solution, the Facebook batch API. Finally, we'll see how we can move slow parts of our code out of the critical path.

Using FQL to Retrieve Information

We've seen how easy it is to use the Facebook API to retrieve data about our users. We have also seen how slow it can be to retrieve more than just a small amount of data. To reduce the need for repeated API calls, Facebook created FQL. FQL is similar to SQL, the Structured Query Language. In fact, the syntax is almost identical. FQL allows us to reduce the number of API calls run by requesting data for multiple users at once.

Let's look at an example FQL query. To get my hometown location, you can use an FQL query like `select hometown_location from user where`

uid=12451752. We'll start by running this query in the API test console.⁵ Select the `fql.query` method from the Method drop-down list. You can enter your query in the query box and click Call Method to see the result. Like SQL, FQL uses the concept of tables of information. Earlier, we wanted information about a user, so we queried the user table.⁶ Although the syntax looks similar, there are a few differences. For example, FQL doesn't allow joins between tables. Additionally, the results of an FQL query vary depending upon the user who runs it. If you aren't my friend on Facebook, you might not be able to see my hometown.

Now that we know a little about FQL, let's look at how we could use it in our application. We built the concept of a dojo into Karate Poke in Section 3.6, *Encouraging Invitations*, on page 66 and Section 6.4, *Spreading by Invitation*, on page 128. We also built a hometown method on our User model. If we were to build a page to display all the members of a dojo and their hometown, that page would need to make an API call for each member of the dojo. That will perform poorly as dojos get larger. We can rework our hometown method using FQL to reduce the number of API calls we'll need to make. We'll start by building an FQL query that will return the hometown for each user. Just like with SQL, we can use the `in` predicate to retrieve information for a list of users:

```
@disciples = current_user.disciples
disciple_ids = @disciples.map(&:facebook_id).join(",")
users=current_user.facebook_session.fql_query(
  "select uid,hometown_location from user "+
  "where uid in (#{disciple_ids})")
```

We start by getting a list of the Facebook IDs for which we want data. Then we build an FQL query and run it by calling the `fql_query` method on a Facebook session. In return, we get a list of `Facebooker::User` objects. These objects will have data for all the fields we requested in our FQL query. If we try to access a field without data, `Facebooker` will make an API request to retrieve that data for us.

Now that we have our list of users, we'll need a way to use this information in our hometown method. Previously, our method created a new `Facebooker::User` object and then retrieved the location from that.

5. Available at <http://developer.facebook.com/tools.php>

6. You can find a list of all the tables in the developer documentation at <http://developer.facebook.com/documentation.php?doc=fql>.

Let's change our hometown method to allow it to use a supplied Facebooker::User instance:

```
def hometown(fb_user)
  fb_user ||= Facebooker::User.new(facebook_id)
  location = fb_user.hometown_location
  text_location =
    "#{location.city} #{location.state}" unless location.blank?
  text_location.blank? ? "an undisclosed location" : text_location
end
```

With that in place, we can just pass the correct Facebooker::User object retrieved from our FQL query to the hometown method. Retrieving the hometown of 40 friends took 28 seconds with the old code. By switching to FQL, that time has decreased to less than two seconds. FQL makes our code run faster, but it also adds complexity. Because of the added complexity, I typically write all my code using the Facebook REST API and convert to FQL only when I really need the performance.

Writing More Complex FQL Queries

I mentioned in the previous section that FQL doesn't support joins. To work around this limitation, FQL queries do support subqueries to retrieve information spanning multiple tables. For instance, we could find all the groups for a user's friends using the following FQL:

```
fql = <<-FQL
  select gid,name
  from group
  where gid in
    (select gid
     from group_member
     where uid in
      (select uid2
       FROM friend
       WHERE uid1 = 12451752)
    )
FQL
groups = current_user.facebook_session.fql_query(fql)
```

Along with writing complex subqueries, FQL also allows you to use functions inside the query. For example, you could retrieve five random friends of a user with the following query:

```
SELECT first_name,last_name,hometown_location
FROM user
WHERE uid IN (SELECT uid2 FROM friend WHERE uid1 = 12451752
              ORDER BY rand() LIMIT 5)
```

FQL provides a really powerful language for retrieving data from Facebook. It provides a speed benefit at the expense of more complex code. It isn't something I use often, but it's a nice tool to have in your belt.

Batching API Calls

Facebook provides a batch request API to perform multiple API calls with only one HTTP request. To use the batch API, you provide Facebook with a JSON-encoded array of request URLs.⁷ Facebook will then run all your requests and return the results for each call.

Instead of going through all this, Facebooker provides a much nicer interface to the batch API. Facebooker allows us to batch calls simply by wrapping them in a call to the `Facebooker::Session#batch` method. For instance, to update a group of users' profiles in a single API call, we could use the following code:

```
facebook_session.batch do
  @users.each do |user|
    update_profile_of(user)
  end
end
```

At the end of the block, a single API request will be sent to update all the profiles. This can significantly decrease the amount of time spent making API calls by reducing the number of HTTP round-trips.

The batch API can do more than just send data; it can also retrieve data. For example, we previously used FQL to retrieve the hometown locations for a list of users. Instead, we could have used the following code:

```
facebook_session.batch do
  @users.each do |user|
    fb_user = Facebooker::User.new(user.facebook_id)
    @hometown_locations << fb_user.hometown_location
  end
end
```

This may seem a little strange. After all, we are adding the user's hometown location to the `@hometown_locations` array inside the block, but we know that only one API call is made at the end of the block.

7. Documentation on the batch API is available at <http://wiki.developers.facebook.com/index.php/Batch.run>.

Facebooker uses some powerful Ruby magic to return a proxy object. A proxy object is an object that pretends to be another object. In this case, the proxy takes the place of the hometown location. When accessed after the end of the block, our proxy objects look just like any other hometown location.

Let's try an example. Here, we use the batch API to retrieve a list of albums for a user. You can see that outside the batch block, `albums` is an array object:

```
>> ses.batch do
?> @albums = ses.user.albums
>> end
=> nil
>> @albums.size
=> 3
```

Since the proxy object doesn't have a value until the end of the block, attempting to access it before then will raise an error, as shown here:

```
>> ses.batch do
?> @albums = ses.user.albums
>> @albums.size
>> end
Facebooker::BatchRequest::UnexecutedRequest: You must execute ...
```

The batch API has some limitations, however. Currently, Facebook limits a batch request to twenty method calls. The Ruby `each_slice` method can be used to segment data into appropriately sized chunks. For instance, if we want to retrieve the albums for an unknown number of users using the batch API, we could use code like this:

```
@users.each_slice(20) do |slice|
  ses.batch do
    slice.each do |user|
      @albums << user.albums
    end
  end
end
```

Although this will reduce the number of API calls we make, it still isn't optimal. For retrieving large amounts of data, FQL is still faster than batched requests.

Additionally, all requests in a batch will share the same session key. This isn't a problem for updating profiles or sending notifications, but it is for publishing feeds. Since each feed item must be published by the acting user, you will be unable to improve performance by batching feeds.

The Importance of Latency with Rails

Request latency is very important to a Rails application. Because each web server process takes a relatively large amount of memory, we are limited to running just ten or fifteen processes on each machine. If our page request takes three seconds to run, that means we will need to run thirty server processes just to handle ten requests per second. It's not unusual for our Facebook applications to have spikes of more than 100 requests per second. To handle that with a three-second response time, we would need 300 processes. That's a lot of hardware!

If we can get our average page load time down to a more reasonable 0.2 seconds, we could handle the same load with only twenty processes. Since some API requests such as profile updates tend to take at least 0.5 seconds to execute, we'll need to find a way to get them out of our request flow.

Move API Calls Out of Line

Even batching API calls doesn't solve all our problems. Although updating twenty profiles in a batch is faster than making twenty requests, it will still take several seconds. While the updates are executing, our user is waiting for a web page to load. If we could move the profile updates out of the request flow, we could get responses back to our users more quickly.

There is no shortage of methods for asynchronous task execution in the Rails world right now.⁸ During the first few months of 2008, I tried just about every system in existence and settled on Starling.⁹ Starling is a persistent message queue written in Ruby. It was created by Twitter to help make its service more resilient. I've used it to process almost 100 asynchronous Facebook requests per second since the beginning of 2008. It is easy to set up and run. In fact, *Advanced Rails Recipes* [Cla08] has a recipe explaining how to use Starling for exactly this purpose.

8. You can see a discussion of some alternatives at <http://nubyonrails.com/articles/about-this-blog-beanstalk-messaging-queue>.

9. Starling is available at <http://rubyforge.org/projects/starling/> or by running `gem install starling`.

Any of these methods will meet our goals. Making API calls asynchronous significantly increases the complexity of our application. Not only will we have more processes to monitor, but we'll also have more points of failure. We'll need to consider what happens when we are receiving new messages faster than we can process them. Even so, for an application supporting millions of users, it can make handling the load much easier.

9.5 Summary

We've looked at a number of ways to help our application scale. By reducing the number of database queries that run for each action, and even bypassing actions when possible, we increased the load our application can handle. By batching API calls or moving them out of the request flow entirely, we decreased the amount of time spent processing each request.

This chapter just scratched the surface of scaling a Rails application. Entire books could be written about this one topic. One of the most important things to focus on when improving the performance of your application is measuring your results. If you aren't measuring performance, you'll never know whether your changes are helping or hurting. You also don't need to do all this optimization before launch. Just be standing by in case your application catches on.

Throughout this book, we've covered a lot of ground. We've seen all the basic parts of a Facebook application. We now have a solid User model in our toolkit that can be reused for other applications. We learned how to use messaging and how to put interesting data into our users' profiles. We even looked at scripting with FBJs and learned how to test our applications.

So, what comes next? Become a fan of this book's Facebook page.¹⁰ You'll get updates about new Facebooker functionality. You can also ask questions of other readers. We want to hear about your great Facebook applications. Share them with the group, and show everyone the cool stuff you've done.

10. You can find it at <http://www.facebook.com/pages/Facebook-Platform-Development-with-Rails/12146405638>.

Bibliography

- [Cla08] Mike Clark. *Advanced Rails Recipes: 84 New Ways to Build Stunning Rails Apps*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2008.
- [HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.
- [Knu74] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, 1974.
- [TH05] David Thomas and David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2005.

Index

A

About page, [95](#)
 Action caching, [175](#)
 action parameter, [41-43](#)
 ActionMailer, [107-108](#)
 Actions, [160-161](#)
 ActiveRecord, 54n, [172](#)
 acts_as_cached method, [171](#)
 after_create method, [132](#)
 Ajax (Asynchronous JavaScript and XML), [151-156](#)
 ajax.ondone method, [155](#)
 ajax.onerror attribute, [155](#)
 ajax.post method, [155](#)
 and_return method, [83](#)
 API key, [32-33](#)
 api_key parameter, [68](#)
 Application authorization, [21](#)
 ApplicationController, [38, 160](#)
 ApplicationHelper module, [108](#)
 Applications, *see* Karate Poke
 application; Rails applications
 Arrays, [97](#)
 assert_facebook_redirect_to method, [77](#)
 Asynchronous communications, [185](#)
 Attack model, [60-63, 81-83](#)
 Attacks controller, [85, 91-93](#)
 Authentication
 Facebook support, [36, 52](#)
 Restful Authentication plug-in, [55](#)

B

battle history
 adding, [62](#)
 building battles page, [91-93](#)
 viewing attack results, [22](#)
 before_create callback, [54, 62, 65](#)
 Belt model, [63-66](#)
 body parameter, [126](#)

Browser cookies, [38](#)
 Buck, Jamis, 159n

C

cache_fu plug-in, [171](#)
 Caching
 action, [175](#)
 conditional assignment operator
 and, [58](#)
 fb:ref tag, [177-179](#)
 fragment, [176](#)
 memcached support, [170-173](#)
 pages, [100, 173-174](#)
 views, [173-176](#)
 Callback URL, [31-37](#)
 candelete parameter, [127](#)
 canpost parameter, [127](#)
 Canvas page, *see* Facebook canvas
 change event, [146](#)
 class_name parameter, [61](#)
 clickrewriteform attribute, [151](#)
 clickrewriteid attribute, [151](#)
 clickrewriteurl attribute, [151](#)
 clicktohide attribute, 151n
 clicktoshow attribute, 151n
 Comment forms
 adding, [124-128](#)
 hiding by default, [144-146](#)
 Comment model, [124-128](#)
 comment_on method, [125](#)
 comment_receiver parameter, [126](#)
 comments_url method, [151](#)
 Conditional assignment operator, [58](#)
 Configuring networks, [29](#)
 Content accessibility, [158-160](#)
 content parameter, [41](#)
 content_for method, 129n
 Controller tests, [75-81](#)
 Cookies, [38](#)

create action, [43](#), [87](#), [97](#)
create_by_facebook_id method, [54n](#)
Cross-site forgery attacks, [38](#)
CSS, adding style with, [103](#)
current_user method, [55](#), [86](#)

D

deliver_profile_update method, [132](#)
Developer application, [27](#)
Dialog box messages, [149–150](#)
Dialog class, [149–150](#)
Digital signatures, [37](#), [76](#)
Discussion boards, [128](#)
div tag, [151](#)
DMZ proxy setup, [30](#)
document.getElementById method, [144](#)
DOM, [143](#)
DRY acronym, [94](#)
Dynamic methods, [54n](#)

E

each_slice method, [183](#)
element.getValue method, [147](#)
element.value method, [147](#)
Emails
 notification via, [109–111](#)
 sending, [107–108](#)
ensure_authenticated_to_facebook
 filter, [117](#), [160](#), [161](#)
.erb files, [86](#)
Error handling, [33](#), [122](#)
expire_cache method, [172](#)
expire_fragment method, [176](#)

F

Facebook API
 accessing outside canvas, [166–169](#)
 batching calls, [182](#)
 Facebooker support, [36](#), [45](#)
 functionality, [69](#)
 HTTP requests, [47](#)
 performance considerations,
 [179–185](#)
 POST request, [68](#)
Facebook canvas
 adding navigation, [93–97](#)
 adding pagination, [102–103](#)
 adding style, [103](#)
 building, [26–27](#)
 building battles page, [91–93](#)

 creating forms, [85–90](#)
 error handling, [33](#)
 hiding content from users, [97–101](#)
 Karate Poke application, [22](#)
 selecting path, [31](#), [37](#)
Facebook Developer application, [27](#)
Facebook helpers, *see* Helper methods
Facebook JavaScript, *see* FBJS
Facebook Markup Language, *see* FBML
Facebook Platform
 accessing outside canvas, [166–169](#)
 application authorization, [21](#)
 canvas page, [22](#), [26–27](#)
 cookie support, [38](#)
 developer site, [18](#)
 handling specific data, [161–164](#)
 messages, [24–25](#)
 profile page, [23](#), [26–27](#)
 REST API, [67–70](#)
 sharing information, [166](#)
 Spamminess metric, [111–112](#)
 terms of service, [53](#), [56](#)
Facebook Profile Publisher, [113](#),
 [119–124](#), [141](#)
Facebook Query Language (FQL),
 [179–185](#)
facebook_delete method, [78](#)
facebook_get method, [77](#), [78](#)
facebook_id parameter, [53](#)
facebook_messages method, [88](#)
facebook_post method, [76](#), [78](#)
facebook_session method, [47](#), [55](#), [57](#)
facebook_templates table, [114](#)
Facebooker
 background, [14–17](#)
 Facebook API support, [36](#), [45](#)
 HTTP requests, [68](#)
 installing, [36](#)
 Publisher interface, [107–108](#)
 Rails class, [76–78](#)
 requiring user login, [38](#)
 website, [19](#)
 see also Session class; User class
fb:board tag, [128](#)
fb:comments tag, [127–128](#)
fb:dashboard tag, [96](#), [97](#)
fb:editor tag, [89](#)
fb:editor-text tag, [89](#)
fb:fbml tag, [40](#), [51](#), [95](#)
fb:if-is-friends-with-viewer tag, [100](#)
fb:if-is-user tag, [99](#), [100](#)

fb:if-user-has-added-app tag, 174
 fb:iframe tag, 164
 fb:is-in-network tag, 100
 fb:js-string tag, 156
 fb:multi-friend-input tag, 86
 fb:multi-friend-selector tag, 41, 50
 fb:name tag

- default links, 92
- helper method support, 51
- information storage and, 56
- page caching, 174
- privacy considerations, 101
- uid parameter, 44
- web browsers and, 161

 fb:narrow tag, 133
 fb:profile-pic tag, 44, 51
 fb:prompt-permission tag, 110
 fb:redirect tag, 77, 174
 fb:ref tag, 177–179
 fb:req-choice tag, 44–45
 fb:request-form tag

- content attribute, 45
- helper method support, 50
- sending feedback, 43, 86
- starting special forms, 40

 fb:success tag, 88
 fb:tab tag, 97
 fb:tab-item tag, 93
 fb:tabs tag, 93
 fb:visible-to tags, 137
 fb:wall tag, 124
 fb:wallpost tag, 124
 fb:wide tag, 133
 fb_about_url method, 95
 fb_action method, 96
 fb_dashboard method, 96
 fb_multi_friend_request method, 49, 129
 fb_name method, 51, 92
 fb_profile_pic method, 51
 fb_req_choice method, 50
 fb_request_form method, 129
 fb_sig parameter, 37, 68
 fb_sig_ajax parameter, 159
 fb_sig_in_canvas parameter, 159
 fb_sig_in_profile_tab parameter, 138
 fb_sig_profile_user parameter, 120, 138, 139
 fb_sig_session_key parameter, 165
 fb_tab_item method, 95
 fb_wall method, 124

fb_wallpost method, 124
 FBJS (Facebook JavaScript)

- Ajax and, 151–156
- overview, 143–152

 FBML (Facebook Markup Language)

- Facebook views, 86
- giving feedback, 43–44
- invitation forms, 40–42, 49–51
- limitations, 153
- making invitations interactive, 44–45
- notification support, 105
- updating profiles, 45–49
- see also* Helper methods; Entries
 - beginning with fb:

 Feedback, invitation form, 43–44
 Feeds, *see* News feeds
 Filters

- before_create, 54, 62, 65
- ensure_authenticated_to_facebook, 117, 160, 161
- spam, 106, 111–112
- style sheets, 103

 Firebug plug-in (Firefox), 144
 Firefox browser, 144, 167
 FlexMock framework, 76, 79–83
 foreign_key parameter, 61
 Forgery attacks, cross-site, 38
 Form fields

- adding, 89
- adding brackets, 97

 form tag, 89
 form_for method, 90
 Fowler, Chad, 16
 Fowler, Martin, 79n
 FQL (Facebook Query Language), 179–185
 fql_query method, 180
 Fragment caching, 176
 Friend selectors, 41
 FunWall application, 98

G

GatewayPorts, 30
 gem install flexmock, 79
 gem install json, 36
 gem install json-pure, 36
 gem install memcache-client, 171
 get_cache method, 171
 GitHub, 36
 Grossenbach, Geoffrey, 39n

Growing Gifts application, [14–16](#), [23](#),
[69](#)

H

h method, [126](#)
Handle refs, [177](#)
Headers, adding, [96](#)
Helper methods
 accessing controller methods, [55](#)
 invitation forms, [50](#)
 testing support, [76–78](#)
 see also Entries beginning with fb_
helper_attr method, [55](#)
Hiding content from users, [97–101](#)
hometown method, [78](#), [83](#), [181](#)
hometown_location method, [83](#)
html tag, [40](#)
HTTP requests, [47](#), [68](#)

I

ids parameter, [43](#), [86](#)
iframe, [164](#)
images parameter, [115](#)
index action, [92](#), [126](#), [174](#)
Index performance, [71](#)
Inline frames, [164](#)
innerHTML attribute, [146–149](#)
Invitations
 cleaning up forms, [49–51](#)
 creating forms, [40–42](#)
 encouraging, [66](#)
 feedback to sender, [43–44](#)
 interactive, [44–45](#)
 notifications and, [106](#)
 rewarding users, [66](#)
 sending invitations, [39](#)
 spreading applications, [128–131](#)
 testing with helpers, [76–78](#)
 viral growth, [46](#)
Invitations controller, [39](#)
invite parameter, [41](#)

J

JavaScript
 accessing Facebook, [166–169](#)
 Ajax and, [151–156](#)
 dialog box messages, [149–150](#)
 FBJS and, [143](#)
 profile support, [149](#)
 Prototype library, [144](#)

JSON (JavaScript Object Notation),
[153](#), [155](#), [182](#)

K

Karate Poke application
 accessing from models, [57–59](#)
 adding navigation, [93–97](#)
 adding pagination, [102–103](#)
 adding style, [103](#)
 application authorization, [21](#)
Attack model, [60–63](#)
Belt model, [63–66](#)
building battles page, [91–93](#)
building web forms, [85–90](#)
caching data, [172](#), [174](#)
canvas page, [22](#), [26–27](#)
configuring Rails, [35–37](#)
functionality, [17](#), [20](#)
hiding content from users, [97–101](#)
marketing, [158–161](#)
messages, [24–25](#)
Move model, [59](#)
profile page, [23](#)
rewarding power users, [63–66](#)
source code, [17](#)
spreading by invitations, [128–131](#)
User model, [52–56](#)
keyup event, [146](#)
Knuth, Donald, [73](#)

L

Latency, [184](#)
LeadersController, [160–161](#)
Leaky Abstraction, [47n](#)
LiveJournal.com, [171](#)
login_url method, [77](#)

M

marketing controller, [159](#)
Matchers, [81](#)
MD5 hash function, [37](#)
memcached system, [170–173](#)
Messages
 asynchronous execution, [185](#)
 as call to action, [108](#)
 dialog box, [149–150](#)
 displaying, [88](#)
 Facebooker requirements, [107–108](#)
 sending, [115](#)
 types supported, [24–25](#)

method parameter, 41
Mock Ajax, 151–153
Mock objects, 78–81, 83
Move model, 59
MySQL, 54

N

name method, 56
Navigation
 adding headers, 96
 adding tab bars, 93–95
 linking to about page, 95
Network configuration, 29
new action, 45
new_instances method, 80
News feeds
 aggregating, 116
 defined, 25
 increasing application visibility,
 117–119
 publishing to, 113–116
 viral growth, 46
Notifications
 defined, 25, 106
 via email, 109–111
 invitations and, 106
 spam filtering, 106, 111–112
 testing, 111
 viral growth, 46
numposts parameter, 127

O

Olson, Rick, 55
oncancel method, 150
onconfirm method, 150
Optimization, premature, 73

P

Page caching, 100, 173–174
Pagination, adding, 102–103
Performance
 adding indexes, 71
 API, 179–185
 caching considerations, 173–176
 Facebook applications, 70
 memcached support, 170–173
 premature optimization, 73
 removing queries, 72–73
Permissions
 application authorization, 21

 requesting from users, 110
Poke application, 35
 see also Karate Poke application
POST request, 68, 127
Profile pages
 adding content, 119–124
 adding tabs, 136–141
 caching information, 26–27, 32
 controlling visibility, 136
 FBJS support, 149
 feed item aggregation, 116
 Karate Poke application, 23
 makeovers, 131–141
 updating, 45–49
Profile Publisher (Facebook), 113,
 119–124, 141
profile_fbml= method, 80, 131
profile_update method, 134
Prototype library (JavaScript), 144
Proxy objects, 183
Proxy servers, 30
Publisher interface (Facebooker),
 107–108

Q

Queries
 FQL, 179–185
 performance considerations, 72–73

R

Rails applications
 action caching, 175
 Ajax support, 152–154
 array support, 97
 browser cookies, 38
 conditional assignment operator, 58
 configuring, 32–33, 35–37
 cross-site forgery attacks, 38
 .erb files, 86
 Facebooker support, 16
 latency and, 184
 matching requests, 159
 page caching, 173–174
 REST support, 39
 returning method, 57
 sending emails, 107–108
Rails class (Facebooker), 76–78
remote_form_for method, 153, 155
render_publisher_error method, 122
render_publisher_interface method,
 120

render_publisher_response method, [122](#)

render_to_string method, [48](#), [120](#)

Requests

- defined, [24](#)
- latency in, [184](#)

requireLogin method, [167](#)

REST (Representational State Transfer)

- Facebook support, [67–70](#)
- fb:comments tag, [127](#)
- overview, [39](#)
- performance considerations, [181](#)

Restful Authentication plug-in, [55](#)

returning method (Rails), [57](#)

RFacebook, [16](#)

Ruby on Rails, *see* Rails applications

S

Safari browser, [167](#)

Scaling applications, [176](#)

Schacht, Keith, [14](#)

Scripting, *see* JavaScript

Secret Key, [32–33](#)

send_as method, [108](#), [115](#)

Send_notification method, [107](#)

Session class (Facebooker)

- batching API calls, [182](#)
- functionality, [57–59](#)
- install_url method, [77](#)
- send_notification method, [107](#)
- setting current session, [70](#)

Session keys, [58](#)

session_key parameter, [53](#), [68](#)

Sessions, sharing, [164–165](#)

set_current_user method, [118](#), [123](#), [141](#)

setInnerFBML method, [148](#), [154](#)

setInnerXHTML method, [148](#), [154](#)

setStyle method, [145](#)

setTextValue method, [148](#), [154](#)

should_receive method, [80](#)

show method, [144](#)

showChoice method, [150](#)

showform parameter, [127](#)

showMessage method, [150](#)

Signatures, [37](#), [76](#)

Social graph, [10](#)

Spam filtering, [106](#), [111–112](#)

Spamminess metric, [111–112](#)

Spolsky, Joel, [47n](#)

ssh command, [29](#)

Starling message queue, [184](#)

Stubs, [79](#), [82](#)

Super Poke application, [35](#)

T

Tab bars, adding, [93–95](#)

template_id, [114](#)

Templates, creating, [113–116](#), [161](#)

Test accounts

- creating, [33–34](#)
- privacy considerations, [100](#)
- sending invitations, [48](#)

Testing

- controller tests, [75–81](#)
- creating test accounts, [33–34](#)
- with mock objects, [78](#), [83](#)
- models, [81–83](#)
- notifications, [111](#)
- Profile Publisher, [141](#)
- sending invitations, [48](#)
- with stubs, [78](#), [82](#)

Twitter application, [166](#), [184](#)

type parameter, [41](#)

U

uid parameter, [44](#)

update action, [162](#)

update_attribute method, [57](#)

update_multiple method, [155](#), [156](#)

update_profile method, [80](#)

url attribute, [45](#)

URL refs, [177](#)

User class (Facebooker)

- building User model, [52–56](#)
- caching and, [171](#)
- for method, [87](#)
- name method, [56](#)
- notification support, [109](#)
- retrieving data, [69](#)

User model, [52–56](#)

User profiles, *see* Profile pages

User_id parameter, [118](#)

V

version attribute, [95](#)

View caching, [173–176](#)

Viral coefficient, [46](#)

Viral growth, [46](#), [176](#)

W

Wall posts, [124](#)

Walls, building, [124](#)

wants_interface method, [122](#)

Web forms, creating, [85-90](#)

Websites

accessing Facebook outside canvas,
[166-169](#)

actions and views, [160-161](#)

handling Facebook-specific data,
[161-164](#)

making content accessible, [158-160](#)
sharing sessions, [164-165](#)

Weirich, Jim, [76, 79](#)

will_paginate method, [102](#)

with matcher, [81](#)

X

X Me poke application, [35](#)

xid parameter, [127](#)

XmlHttpRequest object, [154](#)

Agile Development

Learn what it takes to be pragmatic and agile.

Pragmatic Thinking and Learning

Software development happens in your head. Not in an editor, IDE, or design tool. In this book by Pragmatic Programmer Andy Hunt, you'll learn how our brains are wired, and how to take advantage of your brain's architecture. You'll master new tricks and tips to learn more, faster, and retain more of what you learn.

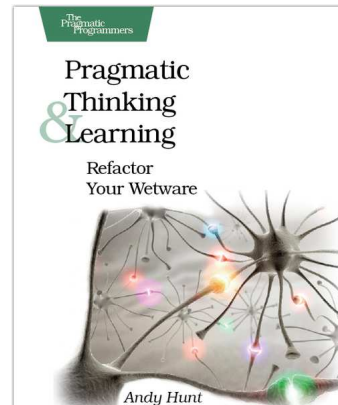
- Use the Dreyfus Model of Skill Acquisition to become more expert
- Leverage the architecture of the brain to strengthen different thinking modes
- Avoid common "known bugs" in your mind
- Learn more deliberately and more effectively
- Manage knowledge more efficiently

Pragmatic Thinking and Learning: Refactor your Wetware

Andy Hunt

(288 pages) ISBN: 978-1-9343560-5-0. \$34.95

<http://pragprog.com/titles/ahpt/>



Practices of an Agile Developer

Agility is all about using feedback to respond to change. Learn how to

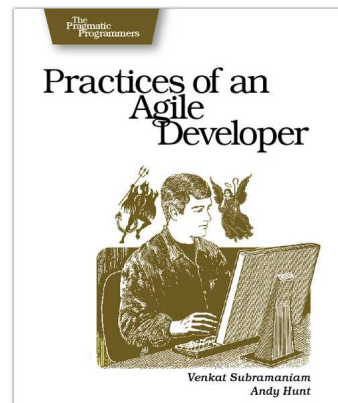
- apply the principles of agility throughout the software development process
- establish and maintain an agile working environment
- deliver what users really want
- use personal agile techniques for better coding and debugging
- use effective collaborative techniques for better teamwork
- move to an agile approach

Practices of an Agile Developer: Working in the Real World

Venkat Subramaniam and Andy Hunt

(189 pages) ISBN: 0-9745140-8-X. \$29.95

<http://pragprog.com/titles/pad/>



More on Agile Projects

More practical advice on tuning and managing projects.

Agile Retrospectives

Mine the experience of your software development team continually throughout the life of the project. Rather than waiting until the end of the project—as with a traditional retrospective, when it's too late to help—agile retrospectives help you adjust to change *today*.

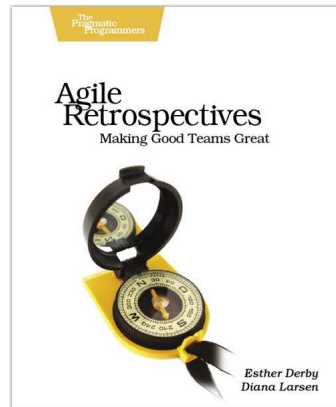
The tools and recipes in this book will help you uncover and solve hidden (and not-so-hidden) problems with your technology, your methodology, and those difficult “people issues” on your team.

Agile Retrospectives: Making Good Teams Great

Esther Derby and Diana Larsen

(170 pages) ISBN: 0-9776166-4-9. \$29.95

<http://pragprog.com/titles/diret>



Manage It!

Manage It! is an award-winning, risk-based guide to making good decisions about how to plan and guide your projects. Author Johanna Rothman shows you how to beg, borrow, and steal from the best methodologies to fit your particular project. You'll find what works best for *you*.

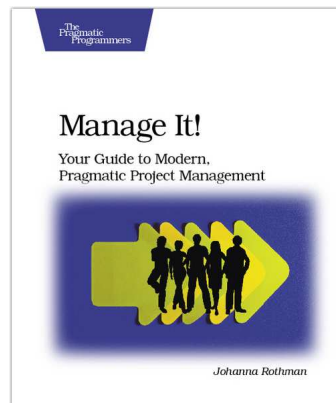
- Learn all about different project lifecycles
- See how to organize a project
- Compare sample project dashboards
- See how to staff a project
- Know when you're done—and what that means.

Your Guide to Modern, Pragmatic Project Management

Johanna Rothman

(360 pages) ISBN: 0-9787392-4-8. \$34.95

<http://pragprog.com/titles/jrpm>



Definitive Ruby and Rails

Help yourself to the definitive reference books for Ruby and Rails.

Programming Ruby 1.9 (The Pickaxe for 1.9)

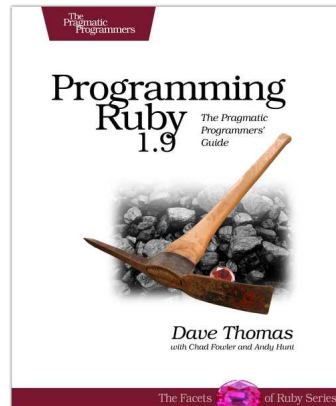
The Pickaxe book, named for the tool on the cover, is the definitive reference to this highly-regarded language.

- Up-to-date and expanded for Ruby version 1.9
- Complete documentation of all the built-in classes, modules, and methods
- Complete descriptions of all standard libraries
- Learn more about Ruby's web tools, unit testing, and programming philosophy

Programming Ruby 1.9: The Pragmatic Programmers' Guide

Dave Thomas with Chad Fowler and Andy Hunt
(992 pages) ISBN: 978-1-9343560-8-1. \$49.95

<http://pragprog.com/titles/ruby3>



Agile Web Development with Rails

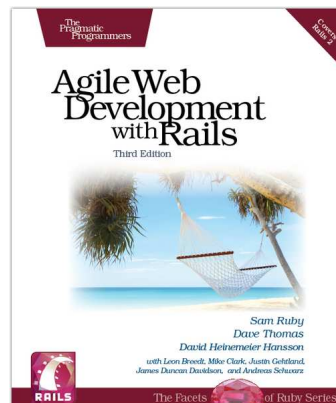
Rails is a full-stack, open-source web framework, with integrated support for unit, functional, and integration testing. It enforces good design principles, consistency of code across your team (and across your organization), and proper release management. This is the newly updated Third Edition, which goes beyond the award winning previous editions with new material covering the latest advances in Rails 2.0.

Agile Web Development with Rails: Third Edition

Sam Ruby, Dave Thomas, and David Heinemeier Hansson, et al.

(784 pages) ISBN: 978-1-9343561-6-6. \$43.95

<http://pragprog.com/titles/rails3>



Ruby and Rails Recipes

Turn your Ruby and Rails development up to eleven.

Advanced Rails Recipes

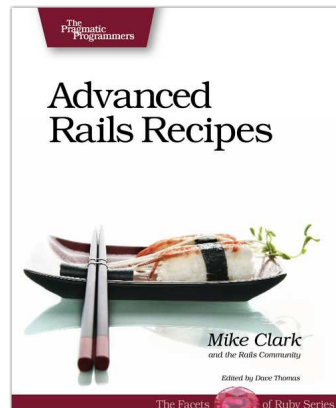
A collection of practical recipes for spicing up your web application without a lot of prep and cleanup. You'll learn how the pros have solved the tough problems using the most up-to-date Rails techniques (including Rails 2.0 features).

Advanced Rails Recipes

Mike Clark

(464 pages) ISBN: 978-0-9787392-2-5. \$38.95

http://pragprog.com/titles/fr_arr



Enterprise Recipes with Ruby and Rails

The 50+ recipes in this book not only show you how to integrate lurking legacy material using Ruby and Ruby on Rails, but also how to create new and highly functional applications in an enterprise environment.

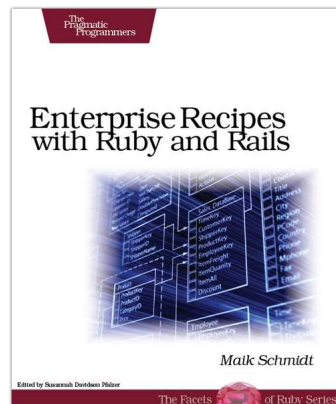
- Work with XML, CSV, fixed length records, and JSON
- Use sockets, SOA, REST and SOAP
- Learn about payment gateways, e-commerce, privacy and security
- Automate tedious enterprise maintenance tasks

Enterprise Recipes with Ruby and Rails

Maik Schmidt

(425 pages) ISBN: 978-1-9343562-3-4. \$38.95

<http://pragprog.com/titles/msenr>



The Real World Web

See how to successfully deploy your Rails project and design your site to be accessible to the widest audience.

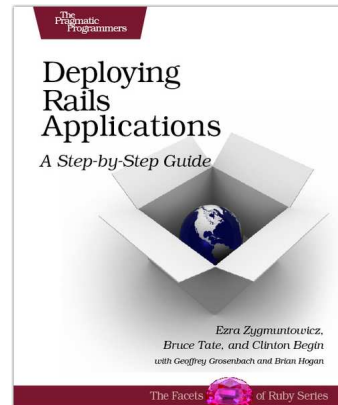
Deploying Rails Applications

Until now, the information you needed to deploy a Ruby on Rails application in a production environment has been fragmented and contradictory. This book changes all of that by providing a consistent, level-headed book containing advice you can trust. You'll get the inside angle from those that have built, deployed, and maintained some of the largest Rails apps in production, anywhere.

Deploying Rails Applications: A Step-by-Step Guide

Ezra Zygmuntowicz, Bruce Tate, and Clinton Begin
(284 pages) ISBN: 978-0-9787392-0-1. \$34.95

http://pragprog.com/titles/fr_deploy



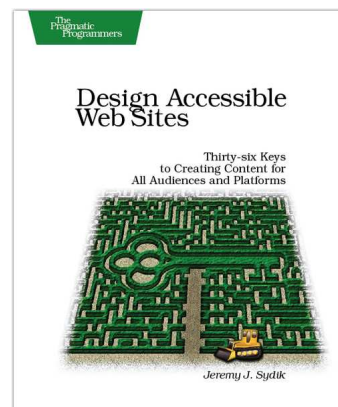
Design Accessible Web Sites

The 2000 U.S. Census revealed that 12% of the population is severely disabled. Sometime in the next two decades, one in five Americans will be older than 65. Section 508 of the Americans with Disabilities Act requires your website to provide *equivalent access* to all potential users. But beyond the law, it is both good manners and good business to make your site accessible to everyone. This book shows you how to design sites that excel for all audiences.

Design Accessible Web Sites: 36 Keys to Creating Content for All Audiences and Platforms

Jeremy Sydik
(304 pages) ISBN: 978-1-9343560-2-9. \$34.95

<http://pragprog.com/titles/jsaccess>



GUI Testing, Ubuntu Tips, and more...

Who says you can't test modern GUIs? We'll show you how.

If you're developing or deploying on Ubuntu, you need this tips and tricks to be more effective and more productive.

For more of our latest titles, please visit www.pragprog.com.

Scripted GUI Testing with Ruby

If you need to automatically test a user interface, this book is for you. Whether it's Windows, a Java platform (including Mac, Linux, and others) or a web app, you'll see how to test it reliably and repeatably.

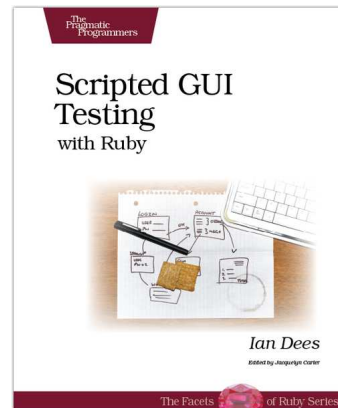
This book is for people who want to get their hands dirty on examples from the real world—and who know that testing can be a joy when the tools don't get in the way. It starts with the mechanics of simulating button pushes and keystrokes, and builds up to writing clear code, organizing tests, and beyond.

Scripted GUI Testing with Ruby

Ian Dees

(192 pages) ISBN: 978-1-9343561-8-0. \$34.95

<http://pragprog.com/titles/idgtr>



Ubuntu Kung Fu

Award-winning Linux author Keir Thomas gets down and dirty with Ubuntu to provide over 300 concise tips that enhance productivity, avoid annoyances, and simply get the most from Ubuntu. You'll find many unique tips here that can't be found anywhere else.

You'll also get a crash course in Ubuntu's flavor of system administration. Whether you're new to Linux or an old hand, you'll find tips to make your day easier.

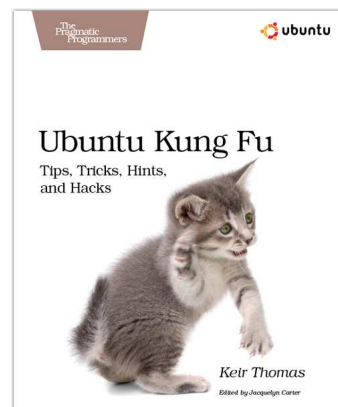
This is the Linux book for the rest of us.

Ubuntu Kung Fu: Tips, Tricks, Hints, and Hacks

Keir Thomas

(400 pages) ISBN: 978-1-9343562-2-7. \$34.95

<http://pragprog.com/titles/ktuk>



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Developing Facebook Platform Applications with Rails' Home Page

<http://pragprog.com/titles/mmfacer>

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/mmfacer.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com