



Community Experience Distilled

Application Development with Parse using iOS SDK

Develop the backend of your applications instantly using
Parse iOS SDK

Bhanu Birani

[PACKT]
PUBLISHING



Community Experience Distilled

Application Development with Parse using iOS SDK

Develop the backend of your applications instantly using
Parse iOS SDK

Bhanu Birani

[PACKT]
PUBLISHING

Table of Contents

[Application Development with Parse using iOS SDK](#)

[Credits](#)

[About the Author](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers and more](#)

[Why Subscribe?](#)

[Free Access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Getting Started with Parse](#)

[Parse backend service](#)

[Advantages of using Parse](#)

[Setting up accounts and applications on Parse](#)

[Integrating Parse in an iOS project](#)

[Setting up Parse](#)

[Summary](#)

[2. Parse Objects and Queries](#)

[Parse objects](#)

[PFObjct](#)

[Operations on Parse objects](#)

[Saving objects](#)

[Fetching objects](#)

[Saving objects offline](#)

- [Updating objects](#)
- [Updating counters](#)
- [Storing data in an array format](#)
- [Deleting objects](#)

[Relations and data types](#)

- [Relations](#)

- [Data types](#)

[Working with queries](#)

- [Using predicates](#)

- [Adding query constraints](#)

[Relational and complex queries](#)

- [Array-based queries](#)

- [String-based queries](#)

- [Relational queries](#)

- [Counting objects](#)

- [Compound queries](#)

- [Caching queries](#)

[Summary](#)

[3. Subclasses and Files](#)

- [Subclassing PFObjct](#)

- [Customizing properties and methods](#)

- [Initializing subclasses](#)

- [Creating queries](#)

- [Handling files](#)

- [PFFile](#)

- [Saving images](#)

- [Tracking progress](#)

[Summary](#)

[4. Parse Analytics](#)

- [Integrating Parse analytics](#)

- [Tracking pages](#)

- [Clearing all badges](#)

- [Reading analytics data](#)

- [Data usage](#)

- [Advantages of analytics](#)

[Summary](#)

[5. Push Notifications](#)

[Setting up push notifications](#)

[Creating an SSL certificate](#)

[Generating certificate requests](#)

[Setting up your App ID](#)

[Creating your provisioning profile](#)

[Configuring your Parse app](#)

[Configuring an iOS application for push notifications](#)

[Installation](#)

[Sending push notifications](#)

[Parse websites](#)

[Parse apps](#)

[Cloud Code](#)

[Sending notifications](#)

[Channels](#)

[Subscribing to channels](#)

[Sending push notifications to channels](#)

[Using advanced targeting](#)

[Sending pushes to queries](#)

[Customizing notifications](#)

[Platform-based targeting](#)

[Receiving push notifications](#)

[Responding to payload](#)

[Summary](#)

[6. Users and Roles](#)

[PFUser](#)

[User login](#)

[Signing up](#)

[Verifying the user](#)

[Logging in](#)

[Saving the user's data](#)

[Creating an anonymous user](#)

[Resetting passwords](#)

[Writing user queries](#)

[Providing security](#)

[Updating the user object](#)

[Securing objects](#)

[Assigning roles](#)

[PFRole](#)

[Securing the role object](#)

[Securing other objects](#)

[Implementing role hierarchy](#)

[Summary](#)

[7. Social Media Integration](#)

[Integrating Facebook](#)

[Prerequisites](#)

[Setting up the application](#)

[Signing up on Facebook](#)

[Publishing on Facebook](#)

[Requesting user data](#)

[Linking users](#)

[Logging out](#)

[Keeping users logged in](#)

[Integrating Twitter](#)

[Setting up Twitter](#)

[Signing up and logging in](#)

[Linking users](#)

[Summary](#)

[8. Using Cloud Functions](#)

[The cloud code](#)

[Installing the command-line tool](#)

[Setting up the cloud code](#)

[Creating the cloud function](#)

[Hosting a website](#)

[Understanding complex functions](#)

[Using cloud on the saved code](#)

[Using cloud on the deleted code](#)

[Summary](#)

[9. Error Handling and Security](#)

[Error handling](#)

[Securing your application data](#)

[Exporting data](#)

[Summary](#)

[Index](#)

Application Development with Parse using iOS SDK

Application Development with Parse using iOS SDK

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2013

Production Reference: 1161013

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK..

ISBN 978-1-78355-033-3

www.packtpub.com

Cover Image by Siddharth Ravishankar
([<sidd.ravishankar@gmail.com>](mailto:sidd.ravishankar@gmail.com))

Credits

Author

Bhanu Birani

Reviewers

Pierce Boggan

Raphael F. P. De Oliveira

Johmel Rey Pintor

Acquisition Editor

Rubal Kaur

Commissioning Editors

Manasi Pandire

Meeta Rajani

Technical Editor

Chandni Maishery

Project Coordinator

Amigya Khurana

Proofreader

Joanna McMahon

Indexer

Hemangini Bari

Production Coordinator

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

About the Author

Bhanu Birani has four years of experience in the software industry and a lifetime association with the technical industry. After years of gaining programming experience in different programming languages, he started developing applications for iOS devices. He started software development during his graduation and was especially interested in learning about the new technologies in the market. He then joined a software company and started developing games, focusing on artificial intelligence development specifically.

I would like to dedicate this book to my family, who gave me the courage and confidence to write it and supported me throughout the entire process. I would also like to send a special thanks to my Mom (Neelu Birani) and Dad (Prakash Birani) for their relentless efforts to assist me in every way imaginable, as well as for helping me keep my life together. Finally, I would like to thank all my friends for sharing my happiness when starting this project, and providing encouragement when it seemed too difficult to complete.

About the Reviewers

Pierce Boggan is a student at Auburn University studying software engineering, with a focus in mobile. Pierce has a passion for software, entrepreneurship, and politics. In the past, he has worked as an intern for Xamarin, a company that allows you to write native cross-platform mobile apps in C#. Pierce now hosts the Xamarin Podcast with Chris Hardy, a bi-monthly discussion of all things Xamarin. In his spare time, you can find him writing about various topics on his blog at pierceboggan.com.

I would like to thank my Savior, Jesus Christ, as well as my friends, my family, and my girlfriend for their support during the entire process.

Raphael F. P. de Oliveira is a passionate software craftsman and an agile and clean code enthusiast. In his spare time he enjoys teaching and, most importantly, loves his life. He teaches iOS development in Rio de Janeiro, Brazil. He has worked on mobile development since 2009 and has published more than 20 apps.

I would like to thank all the great developers that inspire me every day: Martin Fowler, Uncle Bob, Jon Reid, Matt Thompson, among others, and finally my wife, who is always by my side supporting me.

Johmel Rey Pintor is a Computer Engineering graduate from the University of San Carlos in Cebu, Philippines. He learned Objective-C by himself when iOS 4.0 was released. It was a steep learning curve because the aforementioned school did not offer this course. In addition, there was no one to approach when problems occurred, except the Internet. He was also busy with his studies as well as being a working scholar of his school.

It took him about a year and a half of studying Objective-C and Cocoa Touch to finally gain confidence to create applications. Although he has already created working applications, it did not stop him from learning the proper ways of writing readable, simple, and optimized codes by watching the first release of WWDC up to the present.

As Johmel Rey explored different approaches to what his applications could do, he then discovered how to create iOS applications that connect to the Internet, and then sync the data to the device. He first created his backend using PHP that connects to a MySQL database. It was a mess when the foreign key was introduced. Although Johmel Rey prefers to use Parse as a backend service to his applications, and the project worked without a glitch, he couldn't afford to be dependent on Parse.

As NoSQL database gained fame, Johmel Rey thought it was time to learn it. He then learned JavaScript by himself as a prerequisite to node.js, which works perfectly with MongoDB—a NoSQL database behind Parse.

You can follow him on Twitter. His Twitter handle is [@J4hmTor](https://twitter.com/J4hmTor) (<https://twitter.com/J4hmTor>), and on GitHub, <https://github.com/technowar>. You can also check his blog at <http://technowar.github.io>.

First of all, I would like to thank Packt Publishing for giving me this perfect opportunity to share my knowledge on how to use Parse. Secondly, I thank my parents who have been supporting me as I struggled on which career path to pursue. I would also like to thank my girlfriend (as of the moment I am writing this), Roselle Angelie, for being so understanding and loving, and my pet Oddey who has stood by me. Ultimately huge thanks to the big man above.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

Parse is a go-to solution to provide you with a strong and reliable backend service for all the major desktop and mobile platforms such as iOS, Android, Windows Phone 8, JavaScript, OS X, and Windows 8. Parse provides you with the SDK for all platforms, which helps you to provide a strong and reliable backend in no time.

The Parse SDK provides you with an independent way to create your application, without having any dependency on server-side code, and a huge web-service code. Parse emphasizes rapid application development. It significantly accelerates the development process. The Parse platform is easy-to-use, scalable, and reliable.

This book will teach you the fundamentals of how to get started with application development using the Parse SDK. As this book is meant to quickly get you familiar with all the important aspects of the Parse SDK, it will cover lots of subjects very quickly. Throughout the book, we will be dealing with the direct implementations using effective examples and code. This approach will help you to run the example code and integrate the same thing to your project quickly.

Working with the Parse SDK is easy; this will allow you to create your application in no time. The book explains all the topics using code and lots of useful tips explaining every little part.

One of the greatest key points is the Cloud code, where you can place all your code that requires heavy operations. It's useful for multiplatform app development as well, which allows the entire platform to use the same set of code. The Parse Cloud also allows you to host your website, and along with that Parse provides you with prebuilt e-mail verification and reset password mails. Using these services you can send verification mails to a user to verify their e-mail address and reset their passwords respectively.

What this book covers

This book is written to get you familiar with basic fundamentals of application development using the Parse SDK. To fulfill this, the book is organized in easy to understand examples and codes.

[Chapter 1](#), *Getting Started with Parse*, starts by looking at the advantages of using Parse. Then, we will learn about creating an app on Parse and its integration into your project. Finally, we will end the chapter by feeding some example data to the Parse Cloud.

[Chapter 2](#), *Parse Objects and Queries*, explains Parse objects and the operations you can perform on them. We'll learn about relations and data types. We'll also learn about working with queries along with complex and relational queries.

[Chapter 3](#), *Subclasses and Files*, will explain the ways to subclass `PFObject` and its customization to optimize the code. This chapter also explains the ways to handle your files and data.

[Chapter 4](#), *Parse Analytics*, starts by exploring the ways to integrate Parse analytics in your project and reading the analytics data. Finally, we will learn about data usage and its advantages in market analysis.

[Chapter 5](#), *Push Notifications*, starts by explaining the configuration of the Push notification on an Apple developer portal. Then, we will learn about its installation in our project, followed by sending notifications. Then, we will learn about Push sending options. Finally, we will learn about responding to payload (or the data) for the received notifications.

[Chapter 6](#), *Users and Roles*, explains the concept of a user, `PFUser`, along with the various ways you can easily integrate a logging in and sign up workflow in your application. We will also learn about user data management and security. In addition, we will learn about roles to secure application data and restricting its usage to a group of

users only. We will also take a look at the role hierarchy and explore it in detail.

[Chapter 7](#), *Social Media Integration*, discusses the ways to integrate social media into your application. We will explore the ways to integrate Facebook and link it to existing users. Next, we will have a look at Twitter integration with Parse, and explore some ways we can integrate this into our sign up workflow.

[Chapter 8](#), *Using Cloud Functions*, starts by exploring the ways to integrate and host our code on the cloud. Then, we will use the code on mobile applications to minimize the process load on the client side. We will also take a look at a hosting site on Parse. Then, we will dig in and learn some complex function implementations on the Parse Cloud.

[Chapter 9](#), *Error Handling and Security*, will explain the various types of errors that can occur and the ways to handle them. Then we will have a look at the ways to secure data on Parse and how to export our data from Parse.

What you need for this book

You'll need the following things to get started with writing applications for iOS devices using the Parse SDK:

- An Intel-based Macintosh running Leopard (OS X 10.5.3 or later)
- Xcode
- iOS SDK Development Kit
- You must be enrolled as an iPhone developer in order to test the example projects in your device

Who this book is for

If you want to instantly develop an application without being dependent on external web services then this book is for you, as in this book you will learn about Parse, which serves as a backend service. It's a cloud-based platform that takes everything from your head and provides you with the SDK to integrate it into different environments.

This book uses Objective-C as its main language, so some basic knowledge of Objective-C is a must. This book assumes that you understand the fundamentals of object-oriented programming, and programming in general.

This book is designed to get you started using the Parse SDK instantly, so you should be familiar with iPhone/iPad development. The iPhone is a great platform for programming. It looks nice and feels nice. Parse solves your data storage issues and dependency over the web services, and reduces the project development cost and time.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Add the following code inside the `application:didFinishLaunchingWithOptions:` function:"

A block of code is set as follows:

```
PFObject *demoObject = [PFObject
objectWithClassName:@"Demo"]; // 1
[demoObject setObject:@"data value" forKey:@"data
column"]; // 2
[demoObject save]; // 3
```

Any command-line input or output is written as follows:

```
curl -s
https://www.parse.com/downloads/cloud_code/install
er.sh | sudo /bin/bash
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "After creating the application you'll get the **Application ID** and **Client Key**".

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [<feedback@packtpub.com>](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of

our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Getting Started with Parse

Parse is a cloud-based application development platform, which provides you with a scalable and powerful backend service for your application. This platform is available for Windows, iOS, Android, JavaScript, and OS X. Parse provides you with a Plug and Play service for lots of complex components, such as social media integration, push notifications, and data storage with the flexibility of full customization. Parse basically emphasizes rapid application development, using which you can cut down your development time and effort.

In this chapter, we will cover:

- Advantages of using Parse as a backend for our application
- Application setup on the Parse web portal
- Parse SDK integration in our project
- Basic code structure to get started with Parse

Parse backend service

Traditional data-based application development process involves the backend development, which increases the development time and cost of the applications. All the major applications available in the stores are using backend APIs, which itself is a complex development process.

Developers face the following problems while developing applications in a traditional way:

- Large development time and effort
- Adding servers hike the overall product's cost

- Requirement of separate development and production servers for hosting and maintenance
- More prone to bugs
- Application goes down when the server goes down or is under maintenance

To overcome all the drawbacks of traditional application development, developers can leverage Parse to create a powerful backend for their application. Parse is loaded with lots of features that are frequently needed in mobile application development. It reduces the effort, cost, and time for the application's development.

Parse allows you to create your applications without worrying about creating the backend of the application. Using Parse, you can create a flexible and powerful backend without worrying about server management and investing time in writing code for backend servers. Parse provides a complete framework for push notifications, social media integration, analytics, and data storage, with flexibility to code business logic for the backend with Cloud Code.

Advantages of using Parse

There are lots of advantages of using Parse as a backend service. A few of them are listed as follows:

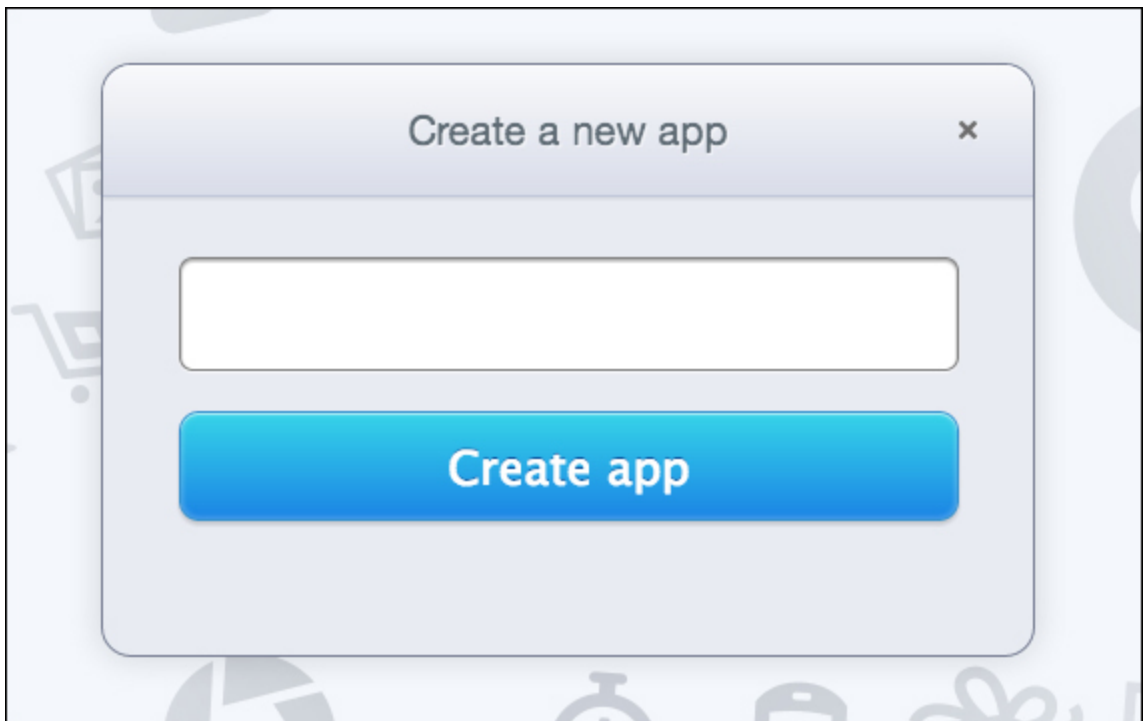
- **Rapid application development:** Parse allows you to rapidly develop applications using native framework components with complete flexibility. It provides a Plug and Play facility for all of the complex components.
- **UX-rich applications:** Parse allows developers to focus more on creating a great user experience and forget about server maintenance and complex infrastructure. You can instantly add push notifications, data storage, social integration (and more!) the moment you integrate the Parse SDK into your app.
- **Powerful data management:** Parse handles everything you need to store data securely and efficiently on the cloud. You can store basic data types, locations, photos, and queries across all of them in just a few lines of code. You can manage, search, filter, and update your content with a web-based data browser.
- **Make your app social:** Connect your application's users via social media sites (such as Facebook and Twitter), with just a few lines of code. Parse takes care of linking accounts across networks, resetting passwords, and keeping everything safe and secure so that you don't have to.
- **Plug and Play push notification:** Parse simplifies the effort of adding real-time push notifications to an application. You can create, send, and target highly effective push notifications via the web-based push console, REST API, or client SDKs. You can send millions of notifications every day; you never have to worry about scaling.
- **Run custom app code:** No need of servers anymore to add rich, custom business logic to your app's backend with Parse's appropriately named Cloud Code. Parse provides you Cloud Code Console with Cloud Modules; which virtually integrate with any third-party service (such as Twilio, MailGun, CrowdFlower, and many more).

- One backend for all: Parse provides native SDKs for iOS, Android, Windows 8, Windows Phone 8, OS X, Unity, Xamarin, and JavaScript; this makes it easy to create beautiful and powerful apps for all devices and environments. From desktop to mobile apps, Parse has an SDK for everyone.

Setting up accounts and applications on Parse

The following steps will help you to set up an account at Parse.com:

1. To start developing your applications using Parse, you need to create your application on Parse.com. You can set up your account by visiting <https://www.parse.com/apps/quickstart>.
2. Follow the instructions and sign up at Parse.com.
3. Create your application by providing a name you wish to integrate with Parse.



4. After creating the application you'll get the **Application ID** and **Client Key**:

Getting started



Your app Test_123 has been successfully created.

Learn how to use Parse in your app using our Quickstart Guide, or start designing your app in the Data Browser.

[Quickstart Guide](#)[Data Browser](#)

App keys

Application ID

USKIP9pNK7X5QEOhRhhkfi4X5iRaXX8i

[Copy](#)

Client Key

00ar000Grww0MINdBklGxj8NS2sxxq4f

[Copy](#)

Javascript Key

G3rAn2Hj1NZ67Q51PMc81lVBuxVG38

[Copy](#)

Windows Key

6kOaqeChA6EjNyrAm6WT2WuUvoY7A`

[Copy](#)

REST API Key

MPPI6xaNSAB1w53kbjYK3Ke7QX5cQD

[Copy](#)

Master Key

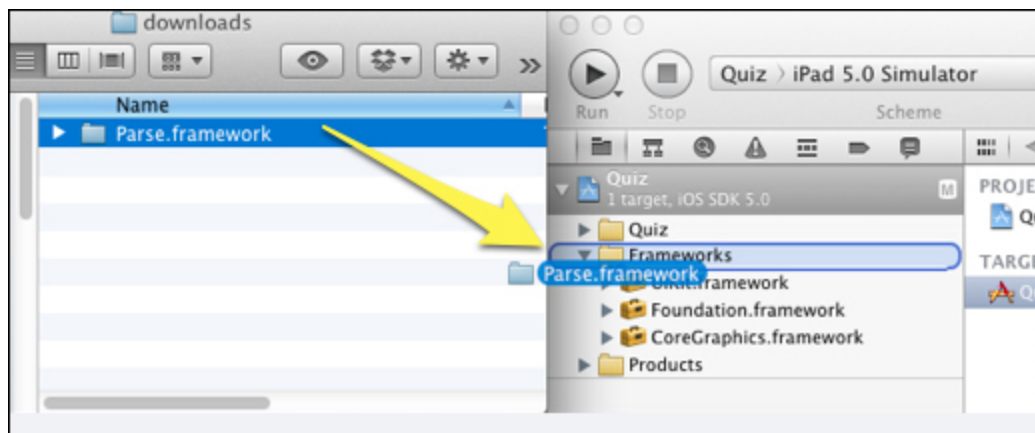
XGnYWmjum1V5T2QVHiD40onfN0UxR

[Copy](#)[Get all your app keys](#)

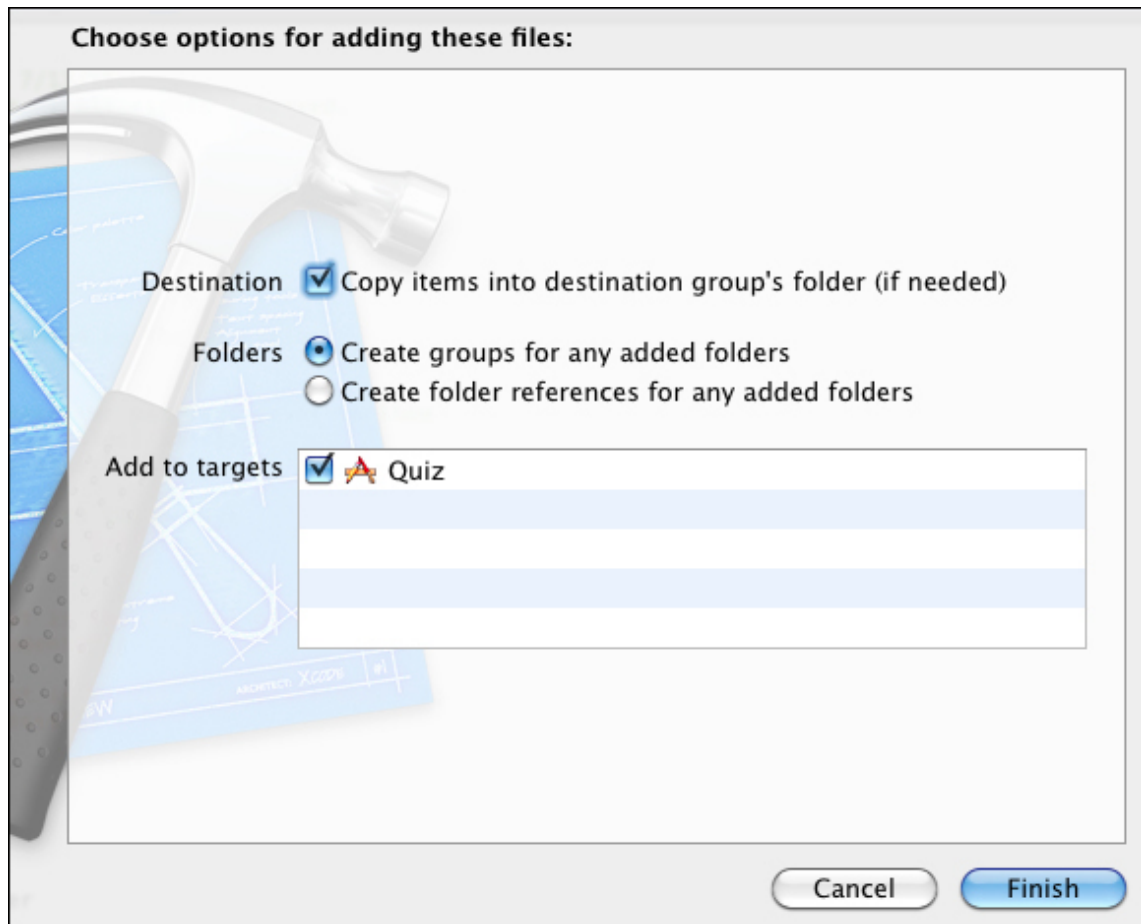
Integrating Parse in an iOS project

The following steps will help you to integrate Parse in your project:

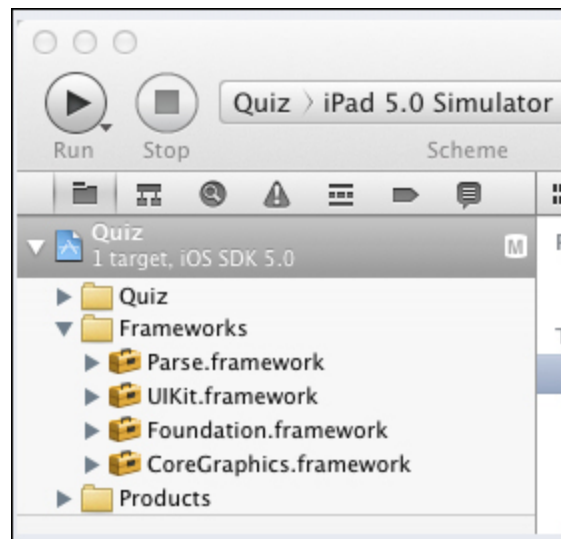
1. Download the Parse iOS SDK.
2. You will need the latest Xcode (v5.0+) and target iOS 4.3 or higher.



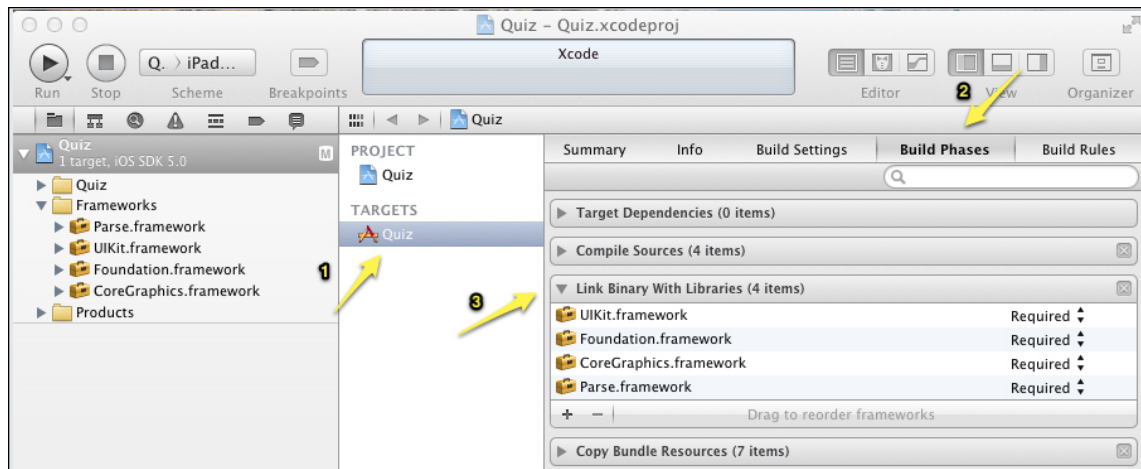
3. Make sure that the **Copy items into destination group's folder** checkbox is checked:



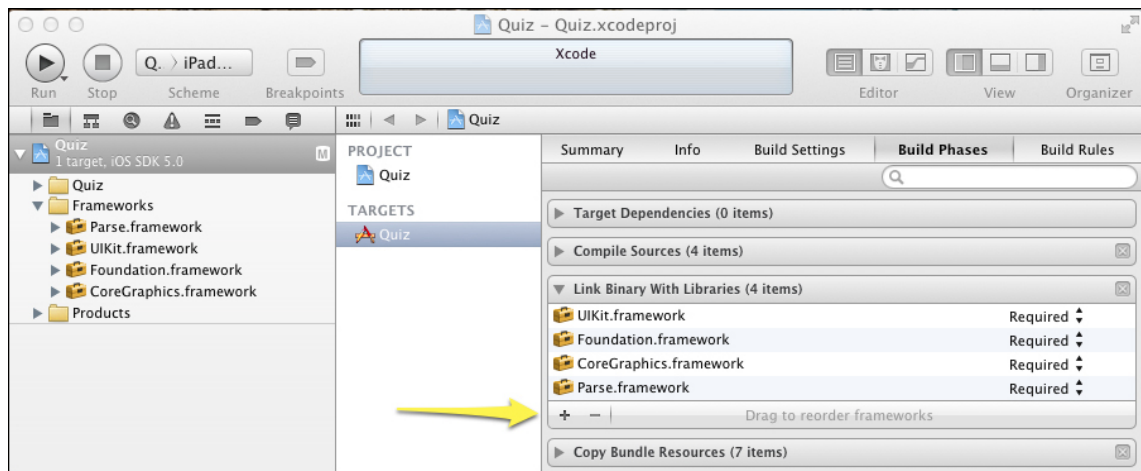
4. After creating the project, it should look like the following screenshot:



5. Add the framework to the project compiled resources. Click on the name of your app under (**Quiz** in our case) **Target | Quiz | Build Phases** tab, and then expand **Link Binary With Libraries** as shown in the following screenshot:

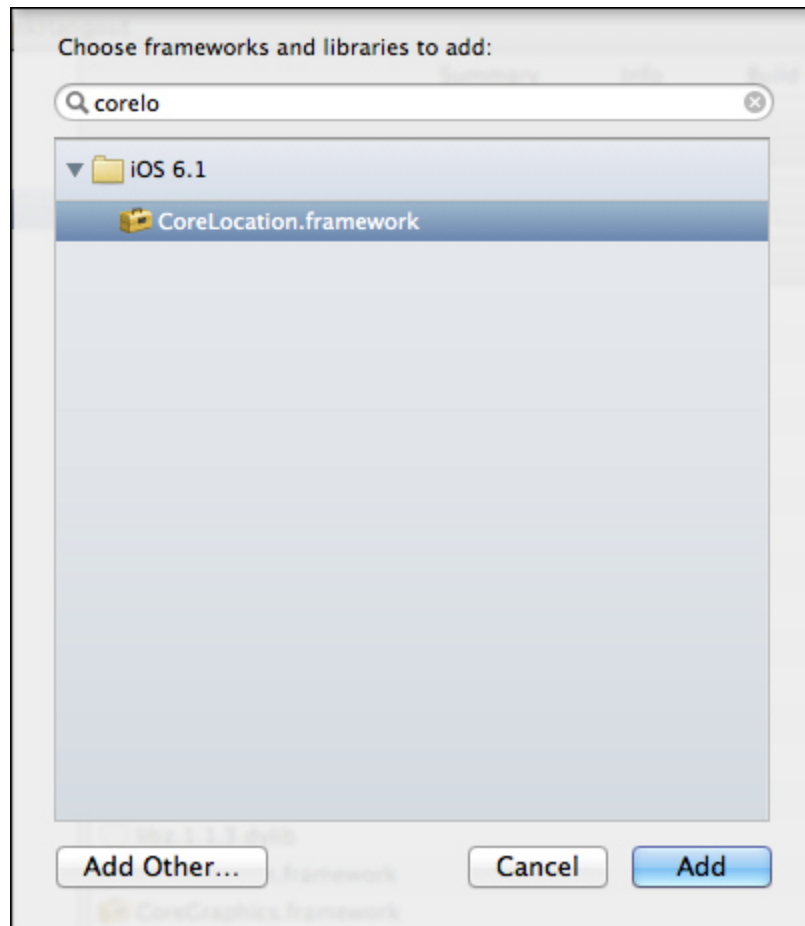


6. Click on the plus button in the bottom left of the **Link Binary With Libraries**:



7. Add the following libraries:

- `CoreLocation.framework`



- CoreGraphics.framework
- libz.1.1.3.dylib
- MobileCoreServices.framework
- QuartzCore.framework
- Security.framework
- StoreKit.framework
- SystemConfiguration.framework
- AudioToolbox.framework
- CFNetwork.framework

Setting up Parse

Now that our project is configured, it's time to integrate Parse into our application's code. Perform the following steps to set up the basic code to get started with Parse:

1. Open the `AppDelegate.m` file and add the following `import` statement on the top of the file:

```
#import <Parse/Parse.h>
```

2. Add the following code inside the `application:didFinishLaunchingWithOptions:` function:

```
[Parse setApplicationId:@"YOUR APP ID"
      clientKey:@"YOUR CLIENT KEY"];
```

3. For tracking statistics when the application opens, add the following line of code:

```
[PFAnalytics
trackAppOpenedWithLaunchOptions:launchOptions]
;
```

4. Compile and run.
5. If you're targeting iOS versions less than 5.0, you'll need to add the **-fobjc-arc** flag to the **Other Linker Flags** entry in your target **Build Settings**:

The screenshot shows the Xcode interface with the 'Build Settings' tab selected for the 'ParseStarterProject' target. The 'Other Linker Flags' setting is highlighted in blue, showing the value '-fobjc-arc'. The 'Path to Link Map File' setting is also visible, showing the value '<Multiple values>'. The 'Perform Single-Object Prelink' setting is set to 'No'.

There you go! You are done with the basic Parse setup in your project. Now you are ready to test your first application on Parse.

Add the following code snippet into your `AppDelegate.m` file at the end of the `application:didFinishLaunchingWithOptions:` method:

```
PFObject *demoObject = [PFObject
objectWithClassName:@"Demo"]; // 1
[demoObject setObject:@"data value" forKey:@"data
column"]; // 2
[demoObject save]; // 3
```

So what exactly are these previous three lines of code doing? Here is the description:

- Line 1: This line simply creates a Parse object with the name "Demo". So, if the class name "Demo" doesn't exist on Parse, it will create a new class with the name "Demo".
- Line 2: This line associates your data with a key within the class. We can understand this as a row-column relationship of the table. The class name is the name of the database table, "Demo" in our case, and the "data column" ID is the name of a column which will hold "data value" as its data.
- Line 3: This line is responsible for actually saving your data on Parse.

Execute your application with the previous lines of code. A new object with the name "Demo" will be sent to the Parse Cloud and then saved.

You can check the results of the code instantly: Go to the Parse web-based **Dashboard**, and then click on your application. Click on the **Data browser** tab and you will observe that under the **Classes** section you have a class name called "**Demo**". When you click on the "**Demo**" class you will observe that the row is created with the following columns:

- `objectId`: This is an automatically generated field by Parse, containing a unique identifier, which will act as a Primary Key for your row.
- `data column`: This column was generated by our code; if you notice you will see that it holds the value we had sent through the code ("`data value`").
- `createdAt`: This is also an auto-generated field which holds the date and time when the row is created.
- `updatedAt`: This is also an auto-generated field, which holds the date and time when the row is updated.
- `ACL`: This field is for the **Access Control List** and is used for data security. We will discuss this field, in detail, in the upcoming chapters.

Finally, we've configured the Parse application on the Parse portal, and configured the project to send and receive data from Parse.

Summary

In this chapter, we learned how to set up an application on Parse, including some basic code and concepts to get us started with Parse. We started by creating an application on Parse. Then, we integrated Parse in your iOS project. Finally, we wrote some code to get started with Parse.

In the next chapter, we will learn about Parse objects and queries.

Chapter 2. Parse Objects and Queries

Parse helps you to manage your complete backend structure for mobile applications and discard the requirement of developing complex server-side code and their maintenance. Parse provides you with the database structures in the form of objects. It saves all your data in the form of objects and will return the same object while retrieving back from Parse.

Every application has a different and specific **Application ID** associated with the **Client Key**, which remains same for all the applications of the same user.

Parse is based on object-oriented principles. All the operations on Parse will be done in the form of objects. Parse saves your data in the form of objects you send, and helps you to fetch the data in the same format again. In this chapter, you will learn about objects and operations that can be performed on Parse objects.

In this chapter, we will learn how to work with Parse objects along with writing queries to set and get data from Parse.

Parse objects

All the data in Parse is saved in the form of **PFObject**. When you fetch any data from Parse by firing a query, the result will be in the form of **PFObject**. The detailed concept of **PFObject** is explained in the following section.

PFObject

Data stored on Parse is in the form of objects and it's developed around PFObject. PFObject can be defined as the key-value (dictionary format) pair of JSON data. The Parse data is schemaless, which means that you don't need to specify ahead of time what keys exist on each PFObject. Parse backend will take care of storing your data simply as a set of whatever key-value pair you want.

Let's say you are tracking the visited count of the username with a user ID using your application. A single PFObject could contain the following code:

```
visitedCount:1122, userName:"Jack Samuel",  
userId:1232333332
```

Parse accepts only string as [key](#). Values can be strings, numbers, Booleans, or even arrays, and dictionaries—anything that can be JSON encoded.

The class name of PFObject is used to distinguish different sorts of data. Let's say you call the [visitedCounts](#) object of the user. Parse recommends you to write your class name [NameYourClassLikeThis](#) and [nameYourKeysLikeThis](#) just to provide readability to the code. As you have seen in the previous example, we have used [visitedCounts](#) to represent the visited count key.

Operations on Parse objects

You can perform save, update, and delete operations on Parse objects. Following is the detailed explanation of the operations that can be performed on Parse objects.

Saving objects

To save your `User` table on the Parse Cloud with additional fields, you need to follow the coding convention similar to the `NSMutableDictionary` method. After updating the data you have to call the `saveInBackground` method to save it on the Parse Cloud. Here is the example that explains how to save additional data on the Parse Cloud:

```
PFObject *userObject = [PFObject currentUser];
[userObject setObject:[NSNumber
numberWithInt:1122] forKey:@"visitedCount"];
[userObject setObject:@"Jack Samuel"
forKey:@"userName"];
[userObject setObject:@"1232333332"
forKey:@"userId"];
[userObject saveInBackground];
```

Just after executing the preceding piece of code, your data is saved on the Parse Cloud. You can check your data in **Data Browser** of your application on Parse. It should be something similar to the following line of code:

```
objectId: "xWMyZ4YEGZ", visitedCount: 1122,
userName: "Jack Samuel", userId: "1232333332",
createdAt:"2011-06-10T18:33:42Z", updatedAt:"2011-
06-10T18:33:42Z"
```

There are two things to note here:

- You don't have to configure or set up a new class called User before running your code. Parse will automatically create the class when it first encounters it.
- There are also a few fields you don't need to specify, those are provided as a convenience:
 - `objectId` is a unique identifier for each saved object.
 - `createdAt` and `updatedAt` represent the time that each object was created and last modified in the Parse Cloud. Each of these fields is filled in by Parse, so they don't exist on `PFObjct` until a save operation has completed.

Note

You can provide additional logic after the success or failure of the callback operation using the `saveInBackgroundWithBlock` or `saveInBackgroundWithTarget:selector:` methods provided by Parse:

```
[userObject
saveInBackgroundWithBlock:^(BOOL
succeeded, NSError *error) {
    if (succeeded)
        NSLog(@"Success");
    else
        NSLog(@"Error %@",error);
}];
```

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>.

If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Fetching objects

To fetch the saved data from the Parse Cloud is even easier than saving data. You can fetch the data from the Parse Cloud in the following way.

You can fetch the complete object from its `objectId` using `PFQuery`. Methods to fetch data from the cloud are asynchronous. You can implement this either by using block-based or callback-based methods provided by Parse:

```
PFQuery *query = [PFQuery
queryWithClassName:@"GameScore"]; // 1
[query getObjectInBackgroundWithId:@"xWMyZ4YEGZ"
block:^(PFObject *gameScore, NSError *error) {
//2
    // Do something with the returned PFObject in
    the gameScore variable.
    int score = [[gameScore objectForKey:@"score"]
intValue];
    NSString *playerName = [gameScore
objectForKey:@"playerName"]; //3
    BOOL cheatMode = [[gameScore
objectForKey:@"cheatMode"] boolValue];
    NSLog(@"%@", gameScore);
}];
// The InBackground methods are asynchronous, so
the code written after this will be executed
// immediately. The codes which are dependent on
the query result should be moved
// inside the completion block above.
```

Let's analyze each line in here, as follows:

- Line 1: It creates a query object pointing to the class name given in the argument.

- Line 2: It calls an asynchronous method on the query object created in line 1 to download the complete object for `objectId`, provided as an argument. As we are using the block-based method, we can provide code inside the block, which will execute on success or failure.
- Line 3: It reads data from `PFObjct` that we got in response to the query.

Parse provides some common values of all Parse objects as properties:

```
NSString *objectId = gameScore.objectId;  
NSDate *updatedAt = gameScore.updatedAt;  
NSDate *createdAt = gameScore.createdAt;
```

To refresh the current Parse object, type:

```
[myObject refresh];
```

This method can be called on any Parse object, which is useful when you want to refresh the data of the object. Let's say you want to re-authenticate a user, so you can call the `refresh` method on the user object to refresh it.

Saving objects offline

Parse provides you with the functions to save your data when the user is offline. So when the user is not connected to the Internet, the data will be saved locally in the objects, and as soon as the user is connected to the Internet, data will be saved automatically on the Parse Cloud. If your application is forcefully closed before establishing the connection, Parse will try again to save the object next time the application is opened. For such operations, Parse provides you with the `saveEventually` method, so that you will not lose any data even when the user is not connected to the Internet. Eventually all calls are executed in the order the request is made. The following code demonstrates the `saveEventually` call:

```
// Create the object.
PFObject *gameScore = [PFObject
objectWithClassName:@"GameScore"];
[gameScore setObject:[NSNumber numberWithInt:1337]
forKey:@"score"];
[gameScore setObject:@"Sean Plott"
forKey:@"playerName"];
[gameScore setObject:[NSNumber numberWithBool:NO]
forKey:@"cheatMode"];
[gameScore saveEventually];
```

Updating objects

For updating objects on Parse, we just need to feed the new data to Parse by calling any of the save functions.

For example, suppose you have `objectId` of the object saved on Parse. We can fetch the corresponding `PFObject` using `PFQuery` from Parse:

```
PFQuery *query = [PFQuery
queryWithClassName:@"GameScore"];

// Retrieve the object by id
[query getObjectInBackgroundWithId:@"xWMYz4YEGZ"
block:^(PFObject *gameScore, NSError *error) {

    // Now let's update it with some new data. In
    this case, only cheatMode and score
    // will get sent to the cloud. playerName
    hasn't changed.
    [gameScore setObject:[NSNumber
numberWithBool:YES] forKey:@"cheatMode"];
    [gameScore setObject:[NSNumber
numberWithInt:1338] forKey:@"score"];
    [gameScore saveInBackground];

}];
```

After executing the previous code, the Parse client will automatically detect the fields that are changed and will send only the changed

fields to Parse for update. Here, we are using the `saveInBackground` method to asynchronously save our data on the Parse Cloud.

Updating counters

In some cases, you may want to update the counters such as in the case of a game score. Then, in those cases, for incrementing the value of the key, you can use the `incrementKey` method, and for decrementing the value, you can use the `decrementKey` method over `PFObjec`t:

```
[gameScore incrementKey:@"score"];  
[gameScore saveInBackground];
```

We can even increase or decrease the value of counters by any specific number by using the following line of code:

```
[gameScore incrementKey:@"score" byAmount:8];
```

The previous line of code will increment the `"score"` key by 8.

Storing data in an array format

To store data in array format, Parse provides several methods for saving the data. To add the data into your existing value of key, you can use the following code snippet. This code will add the `"flying"` object in your `"skills"` key:

```
[gameScore addObject:@"flying" forKey:@"skills"];
```

The following line of code will add the provided array of objects to the existing array on the cloud. (It can have duplicate data for the value of key.):

```
[gameScore addObjectsFromArray:[NSArray  
 arrayWithObjects:@"flying", @"kungfu", nil]
```

```
forKey:@"skills"];
```

In the previous line of code, the data of the array will be appended to the existing cloud data without any check for the uniqueness of the value of the array. In case you want to avoid the duplicate values to be saved on the cloud, you can use the `addUniqueObjectsFromArray` method to save your array. This method will ensure that only unique values are saved on the cloud for the key. The following line of code will add the provided array of objects to the existing array if they are not already added:

```
[gameScore addUniqueObjectsFromArray:[NSArray  
 arrayWithObjects:@"flying", @"kungfu", nil]  
 forKey:@"skills"];  
[gameScore saveInBackground];
```

Deleting objects

The following line of code will help you to delete objects from the Parse Cloud. This method will delete your object asynchronously:

```
[myObject deleteInBackground];
```

You can use the `deleteInBackgroundWithBlock:` method to get a callback after the deletion of the object.

The following code will delete the object in the background and return the success or error message in response:

```
[ myObject deleteInBackgroundWithBlock:^(BOOL  
succeeded, NSError *error) {  
    if (error) {  
        // handle error  
    }  
    if (succeeded) {  
        //code on success  
    }  
}];
```

Relations and data types

Just like our relations in the database, we have multiple tables that have a relationship with each other. Parse provides us with the facility to establish relations in between our Parse classes as well. Relations in Parse are explained in detail in the following section.

Relations

You can add a relationship between two Parse objects. To implement this, we can add one `PFObjct` as the value of another `PFObjct`.

Let's take an example, we have both `Course` and `Student` objects, and we can associate a course with a student by establishing the relationship between both of their classes:

```
// Create course data
PFObjct *myCourse = [PFObjct
objectWithClassName:@"Course"];
[myCourse setObject:@"MBA" forKey:@"courseName"];
[myCourse setObject:@"2 years"
forKey:@"courseDuration"];

// Create student data
PFObjct *studentData = [PFObjct
objectWithClassName:@"Student"];
[studentData setObject:@"Jack Samuel"
forKey:@"name"];
[studentData setObject:[NSNumber numberWithInt:22]
forKey:@"age"];

// Add a relation between the student and course
[studentData setObject:myCourse
forKey:@"registeredCourse"];

// This will save both myPost and myComment
[studentData saveInBackground];
```

Relations between objects can also be achieved by using `objectId`s as well:

```
// Add a relation between the Post with objectId
"1zEcyElZ80" and the comment
[studentData setObject:[PFObject
objectWithoutDataWithClassName:@"Course"
objectId:@"1zEcyElZ80"]
forKey:@"registeredCourse"];
```

Data types

Parse supports all the data type formats such as `NSString`, `NSData`, `NSNumber`, `PFObject`, `NSDate`, `NSNull`, and `NSData`. You can even create nested objects in the form of `NSDictionary` and `NSArray` to store structured data within a single `PFObject`.

Let's create some random variables shown as follows:

```
NSNumber *number = [NSNumber numberWithInt:42];
NSString *string = [NSString
stringWithFormat:@"the number is %i", number];
NSDate *date = [NSDate date];
NSData *data = [@"foo"
dataUsingEncoding:NSUTF8StringEncoding];
NSArray *array = [NSArray arrayWithObjects:string,
number, nil];
NSDictionary *dictionary = [NSDictionary
dictionaryWithObjectsAndKeys:number,
@"number", string, @"string",
nil];
NSNull *null = [NSNull null];

PFObject *bigObject = [PFObject
objectWithClassName:@"BigObject"];
[bigObject setObject:number
forKey:@"myNumber"];
[bigObject setObject:string
forKey:@"myString"];
[bigObject setObject:date forKey:@"myDate"];
[bigObject setObject:data forKey:@"myData"];
[bigObject setObject:array
forKey:@"myArray"];
[bigObject setObject:dictionary
forKey:@"myDictionary"];
[bigObject setObject:null forKey:@"myNull"];
[bigObject saveInBackground];
```

Working with queries

To fetch the saved data from the cloud, we can use the queries to get the required data.

Let's begin with simple queries. We can fetch data from Parse asynchronously by executing these simple lines of code:

```
PFQuery *query = [PFQuery
queryWithClassName:@"GameScore"];

// Retrieve the object by id
[query getObjectInBackgroundWithId:@"xWMyZ4YEGZ"
block:^(PFObject *gameScore, NSError *error) {

    // Now let's update it with some new data. In
    this case, only cheatMode and score
    // will get sent to the cloud. playerName
    hasn't changed.
    [gameScore setObject:[NSNumber
numberWithBool:YES] forKey:@"cheatMode"];
    [gameScore setObject:[NSNumber
numberWithInt:1338] forKey:@"score"];
    [gameScore saveInBackground];

}];
```

Suppose you want to filter the results while fetching from Parse, you can add conditions over the query object and retrieve the `NSArray` of the filtered `PFObject`. You can use the following method to find the data from the Parse Cloud:

```
findObjectsInBackgroundWithBlock:
```

Or you can use:

```
findObjectsInBackgroundWithTarget:selector:
```

Both methods are asynchronous and will execute callback on the main thread. There is one method called `[query findObjects]`; this method will block the thread on which it is executed.

For example, if you want to fetch the list of students registered for an MBA course, we can use the following code:

```
PFQuery *query = [PFQuery
queryWithClassName:@"Student"];
[query whereKey:@"registeredCourse"
equalTo:@"MBA"];
[query findObjectsInBackgroundWithBlock:^(NSArray
*objects, NSError *error) {
    if (!error) {
        // The find succeeded.
        NSLog(@"Successfully retrieved list of %d
students.", objects.count);
    } else {
        // Log details of the failure
        NSLog(@"Error: %@ %@", error, [error
userInfo]);
    }
}];
```

Using predicates

To add constraint to the result of the query we can use the following methods:

```
NSPredicate *predicate = [NSPredicate
predicateWithFormat:@"studentName = 'Jack
Samuel'"];
PFQuery *query = [PFQuery
queryWithClassName:@"Student"
predicate:predicate];
```

The following types of predicate features are supported by Parse:

- You can use comparisons such as `=`, `!=`, `<`, `>`, `<=`, `>=`, and `BETWEEN` with a key and a constant
- Containment predicates, such as `x IN {1, 2, 3}`

- Key-existence predicates, such as `x IN SELF`
- `BEGINSWITH` expressions
- Compound predicates with `AND`, `OR`, and `NOT`
- Sub-queries with `"key IN %@"`, subquery

The following types of predicates are not supported by Parse:

- Aggregate operations, such as `ANY`, `SOME`, `ALL`, or `NONE`
- Regular expressions, such as `LIKE`, `MATCHES`, `CONTAINS`, or `ENDSWITH`
- Predicates comparing one key to another
- Complex predicates with many `ORed` clauses

Adding query constraints

There are various ways to add constraint to the query results. Following are the few ways to implement constraints:

- To find all the students excluding the specified one in the query, Parse provides the `whereKey:notEqualTo:` method to get the selective filtered data from the Parse Cloud. This method is available in various combinations. The following code will get the objects where the student name is not John Simon:

```
[query whereKey:@"studentName"
notEqualTo:@"John Simon"];
```

- We can add multiple constraints to the same query, the resultant objects will match to all the constraints:

```
[query whereKey:@"studentName"
notEqualTo:@"John Simon"];
[query whereKey:@"studentAge" greaterThan:
[NSNumber numberWithInt:18]];
```

- You can limit the result of the query by setting the `limit` of the query object:

```
query.limit = 10;
```

- If you want to get only the first object of the result, you can use `getFirstObject` or `getFirstObjectInBackground`. Both methods will return the first object from the array of objects received in response:

```
PFQuery *query = [PFQuery
queryWithClassName:@"Student"];
[query whereKey:@"studentName"
equalTo:@"John"];
[query
getFirstObjectInBackgroundWithBlock:^(PFOBJECT
*object, NSError *error) {
    if (!object) {
        NSLog(@"The getFirstObject request
failed.");
    } else {
        // The find succeeded.
        NSLog(@"Successfully retrieved the
object.");
    }
}];
```

- You can skip the result by setting `skip`. This can be used for pagination as well:

```
query.skip = 10; // skip the first 10 results
```

- Parse provides you with an easy way of sorting numbers and strings. It allows you to control the order in which the results are returned:

```
// Sorts the results in ascending order by the
name field
[query orderByAscending:@"studentName"];

// Sorts the results in descending order by
the name field
[query orderByDescending:@"studentName"];
```

- You can filter the results by providing the comparisons in the queries:

```
// Fetch list of students who scored < 50
[query whereKey:@"score" lessThan:[NSNumber
 numberWithInt:50]];
```

```
// Fetch list of students who scored <= 50
[query whereKey:@"score" lessThanOrEqualTo:
 [NSNumber numberWithInt:50]];
```

- You can filter the results based on the different values as well. You can provide the array of the values that should appear in the results:

```
// Finds scores from any of Jonathan, Dario,
or Shawn
NSArray *names = [NSArray
 arrayWithObjects:@"Jonathan Walsh",
 @"Dario Wunsch",@"Shawn Simon",nil];
[query whereKey:@"studentName"
 containedIn:names];
    Fetch objects excluding the data
provided in the array.
// Finds scores from anyone who is neither
Jonathan, Dario, nor Shawn
NSArray *names = [NSArray
 arrayWithObjects:@"Jonathan Walsh",@"Dario
 Wunsch",@"Shawn Simon",nil];
[query whereKey:@"studentName"
 notContainedIn:names];
```

Relational and complex queries

You can fire the queries on relational schema to perform operations on the Parse data. Following are the various ways to fetch data from Parse under a relational model.

Array-based queries

If you have keys with data in the array format, you can find objects where array contains value `x`:

```
// Find objects where the array in arrayKey
contains 2.
[query whereKey:@"arrayKey" equalTo:[NSNumber
 numberWithInt:2]];
```

If you want to fetch the data matching multiple values of array, you can use the following code:

```
// Find objects where the array in arrayKey
contains each of the
// elements 2, 3, and 4.
[query whereKey:@"arrayKey"
 containsAllObjectsInArray:@[@2, @3, @4]];
```

String-based queries

You can use `where:hasPrefix:` to add the constraint to the query result:

```
// Finds student name that start with "Stuart".
PFQuery *query = [PFQuery
 queryWithClassName:@"Student"];
[query whereKey:@"name" hasPrefix:@"Stuart"];
```

Relational queries

You can fetch the objects from Parse where the fields match the particular PFObjct. Parse provides a method `whereKey:equalTo:` to fetch data from a query. Let's say you want to fetch a list of students who are enrolled for the MBA course:

```
// Assume PFObjct *myPost was previously created.
PFQuery *query = [PFQuery
queryWithClassName:@"Student"];
[query whereKey:@"course" equalTo:myCourse];

[query findObjectsInBackgroundWithBlock:^(NSArray
*studentList, NSError *error) {
    // list of student enrolled for MBA course
}];
```

In case you want to fetch multiple types of related objects in a single query, Parse provides you with the `includeKey:` method to fetch these kinds of relational results. For example, if you want to fetch the names of the top-scorer students, and you want to fetch their courses at the same time, you can use the following code snippet to get the desired results:

```
PFQuery *query = [PFQuery
queryWithClassName:@"student"];

// Retrieve the top scorer
[query orderByAscending:@"score"];

// Only retrieve the top ten
query.limit = [NSNumber numberWithInt:10];

// Include the course data with each student score
[query includeKey:@"course"];

[query findObjectsInBackgroundWithBlock:^(NSArray
*students, NSError *error) {
    // Top ten scorer Students from all courses
}];
```

Counting objects

There can be some cases in which you just need to know the count of the resultant array. For this you don't need to query the complete data using the `findObjects` method. Parse provides you with the `countObjects` method to fetch the count of the data available on Parse for the query:

```
PFQuery *query = [PFQuery
queryWithClassName:@"Student"];
[query whereKey:@"studentName" equalTo:@"Sean"];
[query countObjectsInBackgroundWithBlock:^(int
count, NSError *error) {
    if (!error) {
        // The count request succeeded. Log the count
    } else {
        // The request failed
    }
}];
```

Compound queries

If you want to fetch the objects as the result of several queries, Parse provides the `orQueryWithSubqueries:` method to fetch these kinds of result.

Let's say you need to get the list of names of the top-scorer and the least-scorer students; the following code snippet gives the desired output:

```
// Query to fetch the list of students with
highest score
PFQuery *highestScorer = [PFQuery
queryWithClassName:@"Student"];
[lotsOfWins whereKey:@"score" greaterThan:
[NSNumber numberWithInt:150]];

// Query to fetch the list of students with lowest
score
PFQuery *lowestScorer = [PFQuery
queryWithClassName:@"Player"];
[fewWins whereKey:@"score" lessThan:[NSNumber
numberWithInt:5]];
```

```

PFQuery *query = [PFQuery orQueryWithSubqueries:
[NSArray
 arrayWithObjects:highestScorer,lowestScorer,nil]];
[query findObjectsInBackgroundWithBlock:^(NSArray
*results, NSError *error) {
    // results contains students with highest and
    lowest score.
}];

```

Caching queries

Caching the data is a process where we store the data received from the remote API calls temporally on a device. So, for the same request, you do not need to fetch the data again and again. Caching improves the efficiency, performance, and reduces load on the server with minimal API calls.

Once data is requested through a query from Parse, it is recommended to cache it, to reduce the number of hits to the Parse Cloud. You can cache the result on the disk. This feature will help you to show data in an offline mode or when the application is just launched and data is not fetched yet. Parse automatically handles flushing the cache when memory is low.

By default, `PFQuery` doesn't cache queries; however, you can enable caching of the `PFQuery` by setting the property `query.cachePolicy`. Let's take an example, if network reachability is not available, you can add an automatic fallback to cached data:

```

PFQuery *query = [PFQuery
queryWithClassName:@"Student"];
query.cachePolicy =
kPFCachePolicyNetworkElseCache;
[query findObjectsInBackgroundWithBlock:^(NSArray
*objects, NSError *error) {
    if (!error) {
        // Results were successfully found, looking
        first on the
        // network and then on disk.
    } else {
        // The network was inaccessible and we have no
        cached data for
    }
}];

```

```

        // this query.
    }
}];

```

Parse provides you with the following caching policies:

<code>kPFCachePolicyIgnoreCache</code>	It is the default cache policy. The query does not load from the cache nor does it save results to the cache.
<code>KPFCachePolicyCacheOnly</code>	The query only loads from the cache, ignoring the network. If there are no cached results, that causes a <code>PLError</code> .
<code>KPFCachePolicyNetworkOnly</code>	It does not load from the cache, but it saves results to the cache.
<code>KPFCachePolicyCacheElseNetwork</code>	It tries to load from the cache, but if the query fails, it loads results from the network. If neither cache nor network succeeds, there is a <code>PLError</code> .
<code>KPFCachePolicyNetworkElseCache</code>	This query first tries to load from the network, but if it fails, it loads results from the cache. If neither network nor cache succeeds, you will get a <code>PLError</code> .

<code>KPFCachePolicyCacheThenNetwork</code>	This query first loads from the cache, then loads from the network. In this case, the callback will be called twice—first with the cached results, then with the network results. Since it returns two results at different times, this cache policy cannot be used synchronously with <code>findObjects</code> .
---	---

Caching behavior can be controlled using the following operations on the cache:

- It helps you to check the cached result for a query:

```
BOOL isInCache = [query hasCachedResult];
```

- It clears cached results for the query object:

```
[query clearCachedResult];
```

- It clears all the cache of the `PFQuery`:

```
[PFQuery clearAllCachedResults];
```

- You can control the time for which the cache result will exist:

```
query.maxCacheAge = 60 * 60 * 24; // One day,  
in seconds.
```

Summary

In this chapter, we explored Parse objects and the way to query the data available on Parse.

We started by exploring Parse objects and the ways to save these objects on the cloud.

Then, we learned about the queries which will help us to fetch the saved data on Parse.

Finally, we saw various ways to implement queries and the constraints by which we can reduce the complexity of the queries.

In the next chapter, we will learn the various ways to save our files on the Parse Cloud, along with the subclassing concepts on Parse objects.

Chapter 3. Subclasses and Files

Parse provides you with the quick Plug and Play approach to get started with your application development. You can save and fetch data on Parse using `PFObjct`, and any data field of your object can be accessed using the `objectForKey` method. To reduce the complexity along with improving readability and scalability to the codebase, you can use subclass for your `PFObjct`. This also helps you to mold the application architecture in model view controller architecture. In this chapter, we will learn how to subclass `PFObjct` along with ways of saving your files on the Parse Cloud.

Subclassing PFObjct

`PFObjct` can be subclassed to provide the flexibility to the code. Basically it helps you to mold the backend code in an object-oriented format to improve the readability and application architecture. It also helps to make your code re-usable. For example, you can save your data on Parse by following the traditional way:

```
// Create/Save course data
PFObjct *myCourse = [PFObjct
objectWithClassName:@"Course"];
[myCourse setObject:@"MBA" forKey:@"courseName"];
[myCourse setObject:@"2 years"
forKey:@"courseDuration"];
[myCourse saveInBackground];
```

After subclassing, you can transform the previous code into the following code:

```
Course *courseObject = [Course object];
courseObject.courseName = @"MBA";
```

```
courseObject.courseDuration = @"2 years";  
[courseObject saveInBackground];
```

By subclassing `PFObjct` to create the `Course` object, you can easily interact with business objects. Additionally, by using the properties in the code, it increases the readability to your codebase, and, as a result, it simplifies your code architecture.

Perform the following steps to create a subclass of `PFObjct`:

1. Create a subclass of `PFObjct` to follow the `PFSubclassing` protocol.
2. Implement the class method `parseClassName`. This method will return the name of the class as `string`, which you would pass to `initWithClassName:`.
3. The implementation file (`.m` file) imports `PFObjct + Subclass`. This class includes all the methods in `PFSubclassing`.
4. In `AppDelegate` before `userInfo` `setApplicationId:clientId:` call the `[YourClass registerSubclass]` method.
5. The following code demonstrates the `Course` subclass of `PFObjct`:

```
// Course.h  
@interface Course : PFObjct<PFSubclassing>  
+ (NSString *)parseClassName;  
@end  
  
// Course.m  
// Import this header to let your Course class  
know that PFObjct privately  
// provides most of the methods for  
PFSubclassing.  
#import <Parse/PFObjct+Subclass.h>  
  
@implementation Course  
+ (NSString *)parseClassName {  
    return @"Course";  
}  
@end  
  
// AppDelegate.m
```

```
#import <Parse/Parse.h>
#import "Course.h"

- (BOOL)application:(UIApplication
*)application
didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {
    [Course registerSubclass];
    [Parse setApplicationId:parseAppId
clientKey:parseClientKey];
}
```

Customizing properties and methods

To encapsulate your custom business logic, you can add custom properties and methods to your `PFObject` subclass. By using the `PFSubclassing` approach, you can simplify and organize your code in one place instead of having it scattered in different classes. `PFObject` supports dynamic synthesis of the properties. You can declare properties as you would normally declare; however, this time you need to use `@dynamic` in place of `@synthesis` in your implementation file. Dynamic properties tell the compiler that the setter and getter methods are not implemented by class, but somewhere else as a superclass. Let's say you need to add a property in your `Course` class. The following code will add the properties to your `Course` class:

```
// Course.h
@interface Course : PFObject<PFSubclassing>
+ (NSString *)parseClassName;
@property (nonatomic, strong) NSString
*courseName;
@property (nonatomic, strong) NSString
*courseDuration;
@end

// Course.m
@dynamic courseName;
@dynamic courseDuration;
```

You can access the `courseName` and `courseDuration` properties using `course.courseName` or `[course courseName]`, whereas if you want to set the data to properties you can use `course.courseName = @"MBA"` or `[course setCourseName:@"MBA"]`.

In case you have `NSNumber` as a data type, you can implement it in the following way:

```
@property BOOL isGameOver;  
@property float amount;
```

While fetching the data you can use `course [object objectForKey:@"isGameOver"]`, which will return `NSNumber`, which can be accessed using `boolValue`. Whereas in case of an amount, you can access it using the `floatValue` method. The dynamic getter methods will automatically convert the `float` or `bool` value, and the dynamic setter methods will wrap all the primitive data type methods under `NSNumber`.

Initializing subclasses

Create new objects using the `object` class method. This will automatically create an auto-released instance of provided types, and handles further subclassing. You can use this `objectWithoutDataWithObjectId:` method for creating the reference of the current object:

```
PFObject *parseObject = [PFObject  
objectWithoutDataWithClassName:@"Course"  
objectId:courseObject.objectId];
```

The previous line of code will create a reference of your existing Parse object; that is `Course` in our case.

Creating queries

You can create a query for the subclass objects using the `query` class method. Parse provides you with the `PFQuery` class, which allows you to fetch and save data on the Parse Cloud. `PFQuery` provides you with lots of methods to set and get data from the Parse Cloud. Let's say you need to create a query for your `Course` class, you need to use the following method to create a query:

```
PFQuery *query = [Course query];
[query whereKey:@"courseName"
isEqualTo:PFUser.currentUser.courseName];
[query findObjectsInBackgroundWithBlock:^(NSArray
*objects, NSError *error) {
    if (!error) {
        Course *firstCourse = [objects
objectAtIndex:0];
        // Your code here
    }
}];
```

The previous code creates a query to fetch your `Course` class data with the `courseName` filter. You will get an array of all the courses subscribed by the current user. This operation will be executed asynchronously as we are executing this on the background thread. PFQuery provides you with variations of methods to fetch and save data on the Parse Cloud.

Handling files

In applications, sometimes we need to store our files and data in the database or backend. The file can be of any type, it can be our image file, video file, audio file or data file. Parse provides us with the Plug and Play feature to save data and files on the cloud.

PFFile

The Parse application can support any type of file such as images, video files, audio files or any other kind of data file, but their size should be less than 10 megabytes. PFFile helps you to store your application-related files on the Parse Cloud.

For saving the data on the cloud, you need to convert your files into NSData. Then you can create a PFFile object from that and pass NSData to the PFFile object to save on the cloud:

```
NSData *data = [@"Hello world!"
dataUsingEncoding:NSUTF8StringEncoding];
PFFile *exampleFile = [PFFile
fileName:@"brochure.txt" data:data];
```

PFFile provides the `fileName: data:` method to save your files on the Parse Cloud. You can save any kind of file on Parse such as images, text files, and data files. You need to convert all your files to NSData before saving them on Parse. In the previous code, we have converted our `Hello world!` string into NSData before saving it to the Parse Cloud. The filename should be in NSString format, in our case, we have used `brochure.txt` as the filename.

In the previous lines of code, we have used the name of the file as `demo.txt`. Remember the following key points:

- You don't need to worry about the filename collision on the cloud, as Parse will automatically take care of these things by

providing a unique identifier for all your files, so you can have multiple files of the same name.

- It's recommended to provide an extension to your data files, as this will help Parse to understand your file format and handle the files appropriately. So while storing images you should use `.png` or `.jpg` as extensions.

Now you can save your file to the cloud. All the save methods are available to `PFFile` as well. You can use any method that fits your requirement:

```
[file saveInBackground];
```

After successfully saving your file, you can associate your file with `PFObj` just as you associate other data:

```
PFObj *parseObject = [PFObj  
objectWithClassName:@"Course"]  
[parseObject setObject:@"MBA"  
forKey:@"courseName"];  
[parseObject setObject:exampleFile  
forKey:@"brochure"];  
[parseObject saveInBackground];
```

The previous lines of code will associate your `brochure` file with the `Course` class. You can fetch the same file from Parse by using the following code:

```
PFFile *fetchObj = [parseObject  
objectForKey:@"brochure"];  
NSData *resumeData = [fetchObj getData];
```

To fetch data from the Parse Cloud you can use the `getData:` method and its variants. Parse provides you with all the variants of the `getData` method just like what we have seen for saving `PFObj` on the cloud.

Saving images

To save your images on the cloud, you have to convert your images into NSData and then you can save that data to the cloud. Let's say you have an image and you have to save it to the cloud, then you should perform the following steps:

```
UIImage *imageObj = [UIImage
imageNamed:@"picture.png"];
NSData *imageData =
UIImagePNGRepresentation(imageObj);
PFFile *imageFile = [PFFile
fileName:@"picture.png" data:imageData];
[imageFile save];

PFObject *userPhoto = [PFObject
objectWithClassName:@"UserPhoto"];
[userPhoto setObject:@"Test Image"
forKey:@"imageName"];
[userPhoto setObject:imageFile
forKey:@"imageFile"];
[userPhoto save];
```

In the previous lines of code, we started by converting an image file into NSData. Then we created a PFFile object with that data as the value for the image key named as `picture.png`. Then we called the `save` function on the PFFile object to save the file. So now this PFFile object can be used as a value for any other Parse object. Next we created an object on the `UserPhoto` class and saved the PFFile object as a value for our `imageFile` key. Then finally we have invoked the `save` method on the `UserPhoto` parse object.

Tracking progress

Parse helps you to track the progress of your uploads and downloads from your PFFile using the following provided methods:

1. Using the following method you can track the progress of the data you are saving on the cloud:

```
saveInBackgroundWithBlock:progressBlock
```

2. Using the following method you can track the progress while fetching data from the cloud:

```
getDataInBackgroundWithBlock:progressBlock
```

Let's take an example to illustrate how to use your application in progress tracking:

```
NSData *data = [@"Progress tracking of files is  
Easy!" dataUsingEncoding:NSUTF8StringEncoding];  
PFFile *file = [PFFile  
fileWithName:@"brochure.txt" data:data];  
[file saveInBackgroundWithBlock:^(BOOL succeeded,  
NSError *error) {  
    // Handle success or failure here ...  
} progressBlock:^(int percentDone) {  
    // Update your progress spinner here.  
    percentDone will be between 0 and 100.  
}];
```

Summary

In this chapter, we've explored the various ways to save our data to Parse. Data can be anything, such as an image file, audio file, video files or simple binary data files.

We started by exploring the steps to subclass our `PFObject`. Then we learned about the `PFFile` class, which allows us to save our data to the cloud. We also learned about saving our data to the cloud and saving images to the cloud.

Finally, we saw various ways to track the progress of our file.

In the next chapter, we will learn about Parse analytics and the various ways to track your application.

Chapter 4. Parse Analytics

Analytics plays an important role in application development. Analytics provides you with data and shows how it's impacting the customer base. Analytics helps organizations and management take vital decisions to move forward and make business plans and strategies. You can get the detailed knowledge of customer behavior using analytics.

Parse provides you with in-built support for analytics. It's a Plug and Play analytics approach to track your application.

Parse analytics will help you to track your application and helps in management decisions to make the application a success. In this chapter, we will learn about the integration of Parse analytics along with the ways to use the data generated by those analytics.

Integrating Parse analytics

To track your application on Parse you need to plug the following code in your `applicationDelegate` file in the `application:didFinishLaunching:` method:

```
[PFAnalytics  
trackAppOpenedWithLaunchOptions:launchOptions];
```

By adding the preceding line of code, you can collect the data on the basis of how frequently your application is opened and what triggered it. This will also include the means by which an application is opened and the effects of push notifications.

If you pass the `launchOptions` parameter as `nil`, the application will only track the standard app opened event. In case your application is running in the background, then your `application:didReceiveRemoteNotification:` method will get

invoked. In this case you need to add the following code in that method:

```
- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary
*)userInfo {
    if (application.applicationState ==
UIApplicationStateActive) {
        // The application was already running.
    } else {
        // The application was just brought from the
background to the foreground,
        // so we consider the app as having been
"opened by a push notification."
        [PFAnalytics
trackAppOpenedWithRemoteNotificationPayload:userInfo];
    }
}
```

Once you receive push notifications for the application, your application badge is incremented, and after launching the application from the push notification, you need to clear the badges. The code to clear badges is in the following section.

Tracking pages

Parse allows you to track your application pages as well. This helps you to understand how frequently your pages are accessed by users. Parse provides you with the `trackEvent:` method to track the pages in your application. Let's say you want to track how frequently your article page is accessed. You can use the following code to track the frequency of this event:

```
[PFAnalytics trackEvent:@"article_page"];
```

There are some cases in which we need to track the page with specific data. For example, in case of searching, it's helpful to track which data is being searched the most. So for tracking the event with

specific data, you can use the `trackEvent:dimensions:` method. This method accepts a dictionary as a parameter, which allows you to pass specific data while tracking the page. Let's say we want to track which article is most frequently read by users. In that case you can use the following code:

```
NSMutableDictionary *dimensions = @{
    // Define the article name
    @"articleName": @"advantages of parse",
    // Provide the topic category
    @"category": @"technology",
};
// Send the dimensions to Parse along with the
'article' event
[PFAnalytics trackEvent:@"article"
dimensions:dimensions];
```

The previous code will help you to track the event along with the custom parameters.

Clearing all badges

To clear all your badges once your application is opened, you need to set the property of the current installation to update the badge number, and ensure to update the badge value on save. To achieve all your scenarios you need to plug the following lines of code:

```
- (void)applicationDidBecomeActive:(UIApplication
*)application {
    PFInstallation *obj = [PFInstallation
currentInstallation];
    if (obj.badge != 0) {
        obj.badge = 0;
        [obj saveEventually];
    }
    // Your code here
}
```

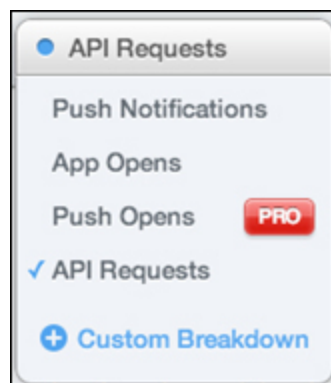
You can access the analytics data from your Parse application dashboard. There you will find options to filter your data as per your

requirements and it will be helpful to track the application based on the tracked criteria.

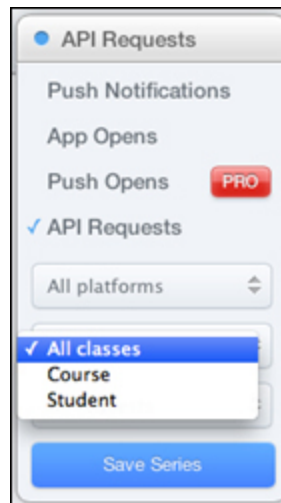
Reading analytics data

Parse analytics provides you with access to various types of data analytics. You can filter your analytics data based on the following filters:

- **Push notifications:** This filter helps you to track the push notifications that were sent by the application to the devices.
- **App opens:** This filter helps you to track how frequently your application is opened. You can even filter your data based on time as well. You can get the data for any particular date, month or week.
- **API requests:** As shown in the following screenshot you can add an **API Requests** filter to the application. This filter helps you to track the number of API requests that are made in a specified time slot:

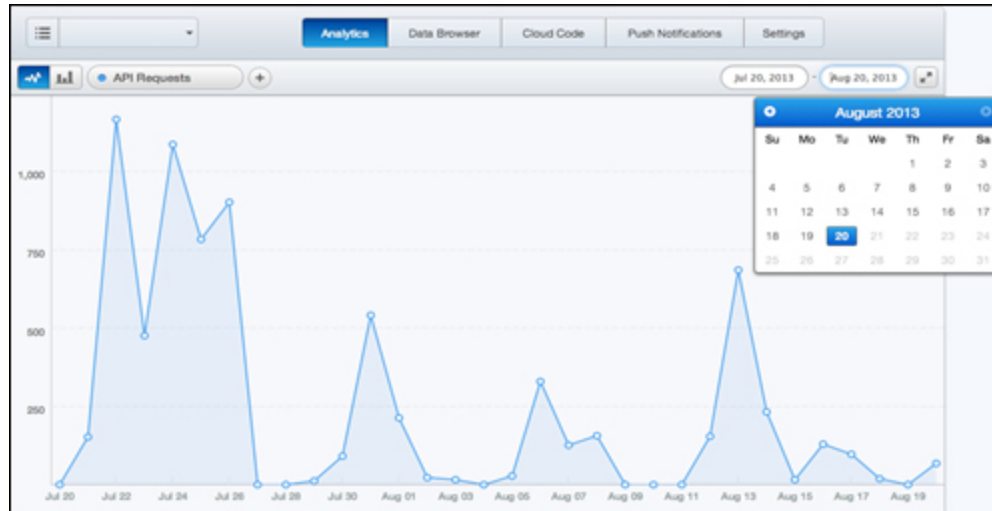


- **Application platforms:** This filter will help you to track the device from which the application has been accessed, such as iOS, Android, Windows, and so on.
- **All classes:** As shown in the following screenshot, this will filter the results based on the classes accessed in our application, and in terms of changes done in that class during our application lifecycle:



- **Request:** This will filter the data based on the type of request made, such as create, get, delete, find, update and find. Based on such requests we can track which operations are performed on the application.

After applying the filter you will have a graph, which will help you to extract how frequently the specified event had occurred:



You can set the time slot for which you need to access the data. The data can be displayed in either graph format or in the bar chart format.

Data usage

Traditionally, companies have used data to forecast trends based on historical evidence. Now you have access to analytics to find the meaning behind the numbers and to differentiate the data based on the associated event. Data charts allow you to make business strategies. For example, the total time the user is spending on their application, and how a small shift can improve their outcomes. Analytics saves time by providing context to the data; this helps us to maximize productivity and prioritize tasks.

You can determine which content sources of your application are getting the most traffic once a visitor starts using the application, along with the frequency of the traffic. This information can be used to find which content is more valuable to the users of the application and will help to develop content more effectively, based on visitor preferences.

Advantages of analytics

There are various advantages of using analytics in mobile applications such as:

- Analytics helps you to track your mobile application to improve the user satisfaction
- It helps you to crunch data and use that data to construct business development models
- Analytics helps you to know your application users and their areas of interest, such as which is the most frequently accessed area of the application

Summary

In this chapter, we've explored Parse analytics and the way to integrate it in our project.

We started by integrating Parse analytics in the project. Then we learned about the ways to read the analytics data from the Parse dashboard. Finally, we have seen some advantages of using analytics in our mobile application.

In the next chapter, we will learn to configure and send push notifications to the users.

Chapter 5. Push Notifications

Push notifications allow you to send messages even when the application is in the background or not running at all. Push notifications increase the user's engagement with the application and keep all the users informed about the application through messages, which in turn generates revenue. Additionally, by having your data and push services interconnected allows you to send push notifications in a selective way, such as *Congrats on completing one year with Packt Publishing*. It's the easiest way to reach all your users. Parse provides you with the push notification services. This chapter will help you to configure push notifications for your application.

Setting up push notifications

Notification helps you in adding real-time messaging to your application. For this, you need to create an SSL certificate and then associate the created certificate with your **App ID** and **Provisioning Profile**.

Creating an SSL certificate

You need to create your **App ID** and associated SSL certificate on the Apple developer portal. The Parse server will use this certificate to send push notifications to the application identified by that **App ID**.

Generating certificate requests

Perform the following steps to generate the certificate and save it to your disk:

1. Open the **Keychain Access** application on your Mac.

2. Navigate to **Keychain Access | Certificate Assistant | Request a Certificate From a Certificate Authority....**
3. In the **Certificate Information** window, enter your e-mail address and name.
4. Select **Save to disk** to download the **.certSigningRequest** file to your desktop.

Setting up your App ID

An App ID is required for all the iOS applications installed on your developer device. You can use your existing App ID, but make sure that it does not have a wildcard character. You can create a new App ID by performing the following steps:

1. Log in to the Apple developer portal (<https://developer.apple.com/membercenter/index.action>) and navigate to **Certificates, Identifiers & Profiles** (<https://developer.apple.com/account/overview.action>).
2. Click on **Identifiers** from the left panel of the portal.
3. Click on the **+** button on the top right-hand side to create a new App ID.
4. Enter the name of the new App ID and select the **Push Notifications** checkbox under the **App Services** section.
5. In the **App ID suffix** section, select **Explicit App ID**. Enter your iOS apps **Bundle ID**. This string should match the Bundle identifier in your iOS app's `info.plist` file.
6. Click on **Continue** followed by **Submit** to finalize your registration.
7. Select your newly created App ID from the list of **iOS App IDs**, then click on **Settings**.
8. Navigate to the **Push Notifications** section. You can download both **Development** and **Production SSL certificate**.
9. Click on **Create Certificate...**, then click on **Continue**.
10. Now you can upload your certificate that you have generated in the previous section and then click on **Generate**.
11. After uploading your certificate, click on **Done** and then download the generated SSL certificate from the **iOS App ID Settings** screen.

12. Install the downloaded certificate in your Keychain by double-clicking on the SSL certificate.
13. In **Keychain Access**, under the **My Certificates** section, select your installed certificate, it should have **Apple Development IOS Push Services:** as a prefix.
14. Right-click on the selected certificate and save it as the **.p12** file format. Do not give any password when prompted.

Creating your provisioning profile

To authenticate your device for the application you need to create a provisioning profile. You need to regenerate your provisioning profile every time you update your App ID. Perform the following steps to create a provisioning profile:

1. Log in to the Apple developer portal and navigate to the **Certificates, Identifiers & Profiles** section.
2. Select **Provisioning Profiles** from the **iOS Apps** section.
3. Click on the **+** button on the top right-hand side of the section, to add a new provisioning profile.
4. Choose **iOS App Development** as the type of your provisioning profile and click on **Continue**.
5. Select the App ID created in the previous section and click on **Continue**.
6. Select your certificate and click on **Continue**.
7. Select the devices from the list of devices on which you want to install and test your application and click on **Continue**.
8. Provide a name to your provisioning profile and click on **Generate**.
9. Download the generated profile and double-click to install it.

This should launch your Xcode's **Organizer** in the **Devices** panel. Your new provisioning profile should appear in the **Provisioning Profile** section of **LIBRARY**. Make sure that the **Status** of the profile is **Valid**.

Configuring your Parse app

To make use of push notification services with Parse, you would have to enable this feature in your Parse app by uploading the Push SSL certificate you generated in the previous sections:

1. Navigate to your Parse app on the Parse **Dashboard** and select the **Settings** tab.
2. On the left-hand side of the settings panel, you will have **Push notifications**. Under the **Apple Push Certificate** header, click on **Select your certificate** and locate your `.p12` certificate you exported from the Keychain in the previous section.

Configuring an iOS application for push notifications

You need to update the following application settings to configure push notifications:

1. Your application bundle identifier should be exactly the same as you have provided on the Apple developer portal while creating the App ID.
2. Update your **Code Signing Identity** field to match the provisioning profile we created in the previous section.

Installation

In all the previous sections, we have configured all that is required to get started with the push notifications. In this section, we will add some code to receive push notifications in our application.

You need to register your device to get the push notifications, add the following code in the app's delegate

`[application:didFinishLaunchingWithOptions:]` method.

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {
    ...
    // Register for push notifications
    [application
registerForRemoteNotificationTypes:

UIRemoteNotificationTypeBadge |
```

```

    UIRemoteNotificationTypeAlert |
    UIRemoteNotificationTypeSound];
    ...
}

```

On successful registration, your callback method

`[application:didRegisterForRemoteNotificationsWithDeviceToken:]` in the app delegate will be executed. We need to configure this method to inform Parse about the new device:

```

- (void)application:(UIApplication *)application
didRegisterForRemoteNotificationsWithDeviceToken:
(NSData *)newDeviceToken {
    // Store the deviceToken in the current
    installation and save it to Parse.
    PFInstallation *currentInstallation =
    [PFInstallation currentInstallation];
    [currentInstallation
    setDeviceTokenFromData:newDeviceToken];
    [currentInstallation saveInBackground];
}

```

You can update the `PFInstallation` just like your `PFOBJECT`. You can add variety of special fields which will help you to manage your devices on Parse. The following are some of the fields and their uses:

- `badges`: You can change this value in `PFInstallation` to update the badge on the app icon. It's recommended to save the changes of badges on the server for future badge increment notifications.
- `channels`: This will store the array of channels that are subscribed to the current device.
- `timeZone`: This property is for the current location of the device. It's updated automatically when we update the `Installation` object on the server. This is a read-only property.
- `deviceType`: This property is for any device type, that can be iOS, Android, winrt, winphone, or dotnet. This is a read-only

property.

- `installationId`: This is a unique ID property for the device used by Parse. This is a read-only property.
- `deviceToken`: This is the Apple-generated token used for iOS devices. This is a read-only property.
- `channelUri`: This is the Microsoft-generated push URI for Windows devices. This is a read-only property.
- `appName`: This is the display name of the client application to which this installation belongs. This is a read-only property.
- `appVersion`: This is the version of string of the client application to which the current installation belongs. This is a read-only property.
- `parseVersion`: This is the version of the Parse SDK that uses this installation. This is a read-only property.
- `appIdentifier`: This is a unique identifier for this installation's client application. In iOS, this is the `Bundle Identifier`. This is a read-only property.

When your application receives notification in foreground mode, you can handle the notification using the following code:

```
- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary
*)userInfo {
    [PFPush handlePush:userInfo];
}
```

After receiving notification, we can pass the notification over to Parse to handle it. It will automatically create a modal alert and will display it with the push message content.

You can now execute your application on your device to make sure that everything is set up correctly. If you are running the app for the first time on your device, you will receive a modal box requesting the permission for sending push notifications to the user.

Sending push notifications

You can send push notifications to devices using the following ways.

Parse websites

Perform the following steps to send notifications through the Parse website:

1. Navigate to your Parse application on Parse.com.
2. Click on the **Push Notifications** tab.
3. You can click on **Send a push** on the top right-hand side of the box.
4. Compose a message in the message box, and broadcast it to **Everyone**. You have other options available to select the device type.
5. Parse allows you to schedule the notifications as well. You can configure this schedule and click on **Send notification** to send the notification.

Parse apps

Usually you can send push notifications through a web console, or by using the REST API, or Cloud Code. However, Parse also allows you to send push notifications from your mobile application as well. Remember, you have enabled the **Client push** as **Yes**. The enabled **Client push** setting allows you to send a notification from one device to another. There are lots of methods available in the `PFPush` class, which allow you to send push notifications through your mobile. You can access all the methods in the API documentation. Here is the example code to demonstrate the push notifications generated from an iOS device:

```
// Create our Installation query
PFQuery *pushQuery = [PFInstallation query];
```

```
[pushQuery whereKey:@"deviceType" equalTo:@"ios"];

// Send push notifications to query
[PFPush
sendPushMessageToQueryInBackground:pushQuery
                                withMessage:@"Hello
World!"];
```

Cloud Code

Cloud Code also allows you to send push notifications using the `Parse.Cloud.afterSave` method, which will be executed after the object is saved successfully on Parse. Cloud can be written in JavaScript. You should be familiar with JavaScript to write Cloud Code. Let's take an example of how to send push notifications after each comment:

```
Parse.Cloud.afterSave("Comment", function(request)
{
    // Our "Comment" class has a "text" key with the
    body of the comment itself
    var commentText = request.object.get('text');

    var pushQuery = new
Parse.Query(Parse.Installation);
    pushQuery.equalTo('deviceType', 'ios');

    Parse.Push.send({
        where: pushQuery, // Set our Installation
query
        data: {
            alert: "New comment: " + commentText
        }
    }, {
        success: function() {
            // Push was successful
        },
        error: function(error) {
            throw "Got an error " + error.code + " : " +
error.message;
        }
    });
});
```

Sending notifications

There are two ways to send push notifications to devices using Parse. They are as follows.

Channels

It's the easiest way to start sending notifications. It's based on the Publish/Subscribe model. For example, if you want to send a notification to the user on every comment on the post he/she created, then you have to create a channel and subscribe the user to that channel, and over every comment you need to send a push for that channel. This will let the user know when anyone comments on his/her post even when the application is not running or is in the background. So for sending the push notifications you need to allow the user to create a channel. A device can subscribe for one or more channels to receive notifications that can be sent to such subscribers. The subscribed channels for `Installation` are stored in the `channels` fields of the `Installation` object.

Subscribing to channels

You can identify a channel by string, which can be a combination of alphanumeric characters, underscores, and dashes. Each installation can subscribe to any number of channels at any time. You can add a channel by using the following code in the `Installation` object:

```
// When users Comment, we subscribe them to that channel.
PFInstallation *currentInstallation =
[PFInstallation currentInstallation];
[currentInstallation addObject:@"Comments"
 forKey:@"channels"];
[currentInstallation saveInBackground];
```

After subscribing to comments, your `Installation` object should look something like this:

```
objectId:yvoZDtAxUR channels:["Comments"]
```

To unsubscribe users from the channel, you can add the following code:

```
// When users indicate they are no longer Giants
fans, we unsubscribe them.
PFInstallation *currentInstallation =
[PFInstallation currentInstallation];
[currentInstallation removeObject:@"Comments"
forKey:@"channels"];
[currentInstallation saveInBackground];
```

To get a list of channels as an array for caching purposes, you can use the following code:

```
NSArray *subscribedChannels = [PFInstallation
currentInstallation].channels;
```

Sending push notifications to channels

To send push notifications to all the users who are subscribed with the `Comments` channel, the following code will send a push to all the subscribed users:

```
// Send a notification to all devices subscribed
to the "Giants" channel.
PFPush *push = [[PFPush alloc] init];
[push setChannel:@"Comments"];
[push setMessage:@"Thats the new comment!"];
[push sendPushInBackground];
```

If you want to send a push to multiple channels at the same time, you can use the following code:

```
NSArray *channels = [NSArray  
arrayWithObjects:@"Comments", @"Mets", nil];  
PFPush *push = [[PFPush alloc] init];  
  
// Be sure to use the plural 'setChannels'.  
[push setChannels:channels];  
[push setMessage:@"Multiple channel comments."];  
[push sendPushInBackground];
```

Using advanced targeting

In advanced push notifications, Parse allows you to embed a query and you can send the result of the query as push notifications to the users. This will help you to send pushes to customize a dynamic segment of users.

You can save the data to the `Installation` object just as any `PFObjct`. Let's say in your application you allow users to like, comment, and post. The following piece of code will help you achieve that:

```
// Store app language and version
PFInstallation *installation = [PFInstallation
currentInstallation];
[installation setObject:[NSNumber
numberWithBool:YES] forKey:@"Like"];
[installation setObject:[NSNumber
numberWithBool:YES] forKey:@"Comment"];
[installation setObject:[NSNumber
numberWithBool:YES] forKey:@"Posts"];
[installation saveInBackground];
```

Sending pushes to queries

You can filter the `Installation` objects by applying queries to the target devices for notifications. This can be achieved using the following code:

```
// Create our Installation query
PFQuery *pushQuery = [PFInstallation query];
[pushQuery whereKey:@"Comments" equalTo:[NSNumber
numberWithBool:YES]];

// Send push notifications to query
PFPush *push = [[PFPush alloc] init];
[push setQuery:pushQuery]; // Set our Installation
query
```

```
[push setMessage:@"This is the test comment."];  
[push sendPushInBackground];
```

We can set the query to `PFPush` before sending the notification. You can also store the data in relationships. Those relationships can also be used in query:

```
// Find users near a given location  
PFQuery *userQuery = [PFUser query];  
[userQuery whereKey:@"location"  
    nearGeoPoint:partyLocation  
    withinMiles:[NSNumber numberWithInt:1]]  
  
// Find devices associated with these users  
PFQuery *pushQuery = [PFInstallation query];  
[pushQuery whereKey:@"user"  
    matchesQuery:userQuery];  
  
// Send push notifications to query  
PFPush *push = [[PFPush alloc] init];  
[push setQuery:pushQuery]; // Set our Installation  
query  
[push setMessage:@"Party tickets free to all  
nearby users!"];  
[push sendPushInBackground];
```

Customizing notifications

You can customize the push notifications data such as sound played, badge number, custom data to send, and expiration of notification.

You can also customize the notification by wrapping all settings in a `NSDictionary` method. Let's take an example of incrementing the badge by one with custom sound and message in the notification:

```
NSDictionary *data = [NSDictionary  
dictionaryWithObjectsAndKeys:  
    @"New comment!", @"alert",  
    @"Increment", @"badge",  
    @"demo.caf", @"sound",  
    nil];  
PFPush *push = [[PFPush alloc] init];  
[push setChannels:[NSArray
```

```
arrayWithObjects:@"Mets", nil]];
[push setData:data];
[push sendPushInBackground];
```

To set the expiration date of the notification, you can use the following example code:

```
// Create date object for tomorrow
NSDateComponents *comps = [[NSDateComponents
alloc] init];
[comps setYear:2013];
[comps setMonth:8];
[comps setDay:27];
NSCalendar *gregorian =[[NSCalendar alloc]
initWithCalendarIdentifier:NSGregorianCalendar];
NSDate *date = [gregorian
dateFromComponents:comps];

// Send push notifications with expiration date
PFPush *push = [[PFPush alloc] init];
[push expireAtDate:date];
[push setQuery:everyoneQuery];
[push setMessage:@"Movie tickets on sale until
August 27th"];
[push sendPushInBackground];
```

Platform-based targeting

You can have the application working on various platforms. If you want to target the push to a specific platform device or operating system, then you can use the following code. This code will send a separate notification for iOS, Android, and Windows apps:

```
PFQuery *query = [PFInstallation query];
[query whereKey:@"channels"
equalTo:@"suitcaseOwners"];

// Notification for Android users
[query whereKey:@"deviceType" equalTo:@"android"];
PFPush *androidPush = [[PFPush alloc] init];
[androidPush setMessage:@"Your suitcase has been
filled with tiny robots!"];
[androidPush setQuery:query];
```

```
[androidPush sendPushInBackground];

// Notification for iOS users
[query whereKey:@"deviceType" equalTo:@"ios"];
PFPush *iOSErrorPush = [[PFPush alloc] init];
[iOSErrorPush setMessage:@"Your suitcase has been
filled with tiny apples!"];
[iOSErrorPush setChannel:@"suitcaseOwners"];
[iOSErrorPush setQuery:query];
[iOSErrorPush sendPushInBackground];

// Notification for Windows 8 users
[query whereKey:@"deviceType" equalTo:@"winrt"];
PFPush *winPush = [[PFPush alloc] init];
[winPush setMessage:@"Your suitcase has been
filled with tiny glass!"];
[winPush setQuery:query];
[winPush sendPushInBackground];

// Notification for Windows 8 users
[query whereKey:@"deviceType"
equalTo:@"winphone"];
PFPush *winPush = [[PFPush alloc] init];
[winPush setMessage:@"Your suitcase is very hip;
very metro."];
[winPush setQuery:query];
[winPush sendPushInBackground];
```

Receiving push notifications

When you receive a customized notification, you can retrieve the associated data from your notification. This data can help in modifying the behavior of the application as per the notification. For example, if the notification is sent because of the comment on the post, you can directly open the post for that comment by reading the push notification data.

Apple has a package size restriction, so it is recommended to keep your data size as small as possible:

```
NSDictionary *data = @{
    @"alert": @"James commented on your photo!",
    @"p": @"vmRZXZ1Dvo" // Photo's object id
};
PFPush *push = [[PFPush alloc] init];
[push setQuery:photoOwnerQuery];
[push setData:data];
[push sendPushInBackground];
```

Responding to payload

If your application is launched using the notification, you can access the data from the `[application:didFinishLaunchingWithOptions:]` method through the `launchOptions` dictionary:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {
    . . .
    // Extract the notification data
    NSDictionary *notificationPayload =
launchOptions[UIApplicationLaunchOptionsRemoteNoti
ficationKey];

    // Create a pointer to the Post object
    NSString *postId = [notificationPayload
objectForKey:@"p"];
    PFObject *targetPost = [PFObject
objectWithoutDataWithClassName:@"Post"objectId:pos
tId];

    // Fetch photo object
    [targetPhoto
fetchIfNeededInBackgroundWithBlock:^(PFObject
*object, NSError *error) {
        // Show post view controller
        if (!error && [PFUser currentUser]) {
            PostVC *viewController = [[PostVC alloc]
initWithPost:object];
            [self.navigationController
pushViewController:viewController animated:YES];
        }
    }];
}
```

If your application is already running in foreground mode, the data can be fetched from the

`[application:didRegisterForRemoteNotificationsWithDeviceToken:]` method in the `userInfo` dictionary:

```

- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary
*)userInfo {
    // Create empty post object
    NSString *postId = [userInfo objectForKey:@"p"];
    PFObject *targetPost = [PFObject
objectWithoutDataWithClassName:@"Post"objectId:pos
tId];

    // Fetch photo object
    [targetPhoto
fetchIfNeededInBackgroundWithBlock:^(PFObject
*object, NSError *error) {
        // Show photo view controller
        if (!error && [PFUser currentUser]) {
            PostVC *viewController = [[PostVC alloc]
initWithPost:object];
            [self.navigationController
pushViewController:viewController animated:YES];
        }
    }];
}

```

Summary

In this chapter, we explored the setup and installation of push notifications in our application.

We started by setting up push notifications on the Apple developer portal and necessary installation in codebase.

Then, we learned about sending the push notifications and their scheduling.

Finally, we saw the ways to receive payload from the notifications.

In the next chapter, we will learn about Parse users and their roles.

Chapter 6. Users and Roles

Most of the applications these days are handling the user accounts and they help users to access their data securely. Parse provides you with the `PFUser` class specifically wrapped with all the functionality required to handle the user's data. All the applications usually contain some sort of users. With Parse and the `PFUser` class, you can easily integrate functionality for authenticating and storing data about users within your application. This class helps you to add the user account functionality in your application.

`PFUser` inherits from `PFObjct`, thus provides you with all the features of a standard `PFObjct`, such as the flexible schema for data feeding, and the dictionary-based key value relationship. `PFUser` has access to all the methods of `PFObjct`, in addition to that, `PFUser` has some additional methods which help to manage account information precisely.

In this chapter, you will learn about handling users in detail, and along with that, you will learn to provide roles to the users to limit the access to the data to all users.

PFUser

This class allows you to manage user accounts on Parse. It has several properties that differentiate it from `PFObjct`, which are as follows:

- `username`: Username for the user (mandatory)
- `password`: Password for the user (mandatory while signup)
- `email`: E-mail ID of the user (optional)

These properties will internally call the `setObject:forKey:` method, so you don't need to call this externally. We will explore these fields in detail in the upcoming topics.

All the `PFUser` objects will be stored in the `User` class. You can access the `User` class in your application's **Data Browser**. This class also allows you to add custom fields to users, such as adding the phone numbers and such relevant details.

User login

In most of the application, the first step is to have an effective login process. Parse provides you with a stable Plug and Play login setup. You can use various login processes to validate users, such as Facebook, Twitter, or the e-mail address of the user. This process is explained in detail in the following topics.

Signing up

You can use the following code to illustrate a way to allow the users to sign up using an e-mail ID and password:

```
- (void)signUp {
    PFUser *user = [PFUser user];
    user.username = @"demo name";
    user.password = @"demo pass";
    user.email = @"demo@example.com";

    // other fields can be set just like with
    PFObject
    [user setObject:@"user display name"
    forKey:@"displayName"];

    [user signUpInBackgroundWithBlock:^(BOOL
succeeded, NSError *error) {
        if (!error) {
            // Successfully signed up
        } else {
            NSString *errorString = [[error
userInfo] objectForKey:@"error"];
            // Show the errorString somewhere and
            let the user try again.
        }
    }];
}
```

The `signUpInBackgroundWithBlock:` method will asynchronously (recommended) create a new user on your Parse application. This method wraps up the basic validations, such as e-mail

authentication, and uniqueness of username and e-mail ID. Parse saves all your passwords on the cloud in hashes. You can save the user's e-mail ID as its username as well.

There are various flavors of the `signUp` methods, which are available on Parse. You can access all these methods from the list of methods in the API docs.

Verifying the user

Parse also enables you to verify the e-mail ID of the user. After verification of an e-mail ID, the `emailVerified` key will be added by the `PFUser` class. This field contains false values and it updates to true as soon as the user has verified his/her e-mail address through their IDs.

Logging in

After signing up, the user will log in to his/her account for accessing the application. For allowing the user to log in to the application, you can use the `logInWithUsernameInBackground:password:` class method of the `PFUser` class:

```
[PFUser logInWithUsernameInBackground:@"demoName"
password:@"demopass"
block:^(PFUser *user, NSError *error) {
    if (user) {
        // Successful login.
    } else {
        // The login failed. Check error to see
        why.
    }
}];
```

After successfully login, you can access the `PFUser` object, which will hold all the user-related data.

Saving the user's data

It is cumbersome for the users of the application to log in every time they open the application. To avoid such cases, you need to save your user data using cache. Saving user's details will allow users to use the application without passing through the login process every time. The `PFUser` class provides an automatic caching policy, and the cached object is saved in the `currentUser` object:

```
PFUser *currentUser = [PFUser currentUser];
if (currentUser) {
    // do stuff with the user
} else {
    // show the signup or login screen
}
```

You can reset the current user by calling the `logout` method on the `PFUser` class:

```
[PFUser logout];
```

Creating an anonymous user

You can allow users to use your application without forcing them to log in to your application. An anonymous user can be created without a username and password, with all the features of `PFUser`. You can convert anonymous users to normal users anytime by providing them with a Facebook, Twitter, or e-mail ID login.

The following code demonstrates a way to create an anonymous user:

```
[PFAnonymousUtils loginWithBlock:^(PFUser *user,
NSError *error) {
    if (error) {
        NSLog(@"Anonymous login failed.");
    } else {
        NSLog(@"Anonymous user logged in.");
    }
}];
```

All the data of an anonymous user will be inaccessible after logout. However, if the user registers itself as a normal user, all the data will be persistent and can be accessed by the user.

Parse provides methods to check whether the user is linked or not. The following code will demonstrate a way to check whether the user is anonymous:

```
if ([PFAnonymousUtils isLinkedWithUser:[PFUser
currentUser]]) {
    [self enableSignUpButton];
} else {
    [self enableLogoutButton];
}
```

Resetting passwords

Often users forget their username and password when they start to log in to your application. In such cases, you need to provide the user with a way to reset their password. Parse provides you with the `requestPasswordResetForEmailInBackground:` method to securely reset the user password.

The following code will illustrate the use of the reset password function:

```
[PFUser
requestPasswordResetForEmailInBackground:@"demo@example.com"];
```

The reset password link will be sent to the provided e-mail ID, and from there the user can enter the new password for the application.

Writing user queries

The following code will demonstrate the way to write query on the `PFUser` class:

```
PFQuery *query = [PFUser query];
[query whereKey:@"gender" equalTo:@"female"]; //
find all the women
NSArray *girls = [query findObjects];
```

The previous code will provide you with all the female users in the output.

Providing security

While you are saving the data of the user, it's important that all the user data should be secure. User data and associated information are significant and they should be saved securely. Parse allows you to save your data securely on the cloud.

Updating the user object

The `PFUser` class is already secured. Data saved in `PFUser` can be updated only by that user. Moreover, you cannot call any `save` or `delete` methods on the `PFUser` class. The only methods that can update the data on the `PFUser` class are the `logIn` and `signUp` methods. This ensures that only the user can update their data on the cloud:

```
The following code will illustrate the security of
the user on cloud.
PFUser *user = [PFUser
logInWithUsername:@"my_username"
password:@"my_password"];
user.username = "my_new_username"; // attempt to
change username
[user save]; // This succeeds, since the user was
authenticated on the device

// Get the user from a non-authenticated method
PFQuery *query = [PFUser query];
PFUser *userAgain = (PFUser *)[query
getObjectWithId:user.objectId];

userAgain.username = "another_username";

// This will throw an exception, since the PFUser
is not authenticated
[userAgain save];
```

To fetch the authentic copy of the user object, use the `currentUser` method, you can check the authenticity of the user by using the

`isAuthenticated` Boolean property.

Securing objects

You can use the `PFUser`'s security model for other objects as well. You can specify the read and write permissions on the objects saved on Parse. You can allow specific group users, which can read the object, and other groups of users, which can modify the object. For security, each object is provided with the access control list, implemented by the `PFACL` class.

The following code will illustrate a way to set the privacy of the object of the current user on Parse:

```
PFObject *privateNote = [PFObject
objectWithClassName:@"Note"];
[privateNote setObject:@"This note is private!"
 forKey:@"content"];
privateNote.ACL = [PFACL ACLWithUser:[PFUser
currentUser]];
[privateNote saveInBackground];
```

The `ACLWithUser` method helps you to generate the ACL that limit the access of the object to the current user only. The ACL is updated after calling the `save` method. The protected data will be accessible to users on the devices on which they are logged in.

You can update the permissions of your Parse objects, respective to users. The following two methods will help you to update the permissions of your Parse objects:

- To provide read access use `setReadAccess:forUser:`
- To provide write access use `setWriteAccess:forUser:`

The following code will illustrate a way to set read and write access:

```
PFObject *groupMessage = [PFObject
objectWithClassName:@"Message"];
PFACL *groupACL = [PFACL ACL];
```

```
// userList is an NSArray with the users we are
sending this message to.
for (PFUser *user in userList) {
    [groupACL setReadAccess:YES forUser:user];
    [groupACL setWriteAccess:YES forUser:user];
}

groupMessage.ACL = groupACL;
[groupMessage saveInBackground];
```

You can also grant read/write permission to all users using the `setPublicReadAccess:` and `setPublicWriteAccess:` methods.

To provide a common access control list for all objects of your application, you can change the default ACLs:

```
PFACL *defaultACL = [PFACL ACL];
[defaultACL setPublicReadAccess:YES];
[PFACL setDefaultACL:defaultACL
withAccessForCurrentUser:YES];
```

In the previous code, we are updating the default ACL to provide public access to all the objects.

Assigning roles

As your application grows, ACL becomes insufficient. You need to be more constrained and have a better approach to control the user access. Parse provides you with the roles to solve the problem. Roles allow you to create logical groups of users. Role is an object that contains users and other roles. Updating the permission of any role will update the permission of all the associated users in that role.

For example, you can divide your application users into Administrators, Standard Users, and Guests roles. Altering the permissions of any role will update the permission of all the users belonging to that group. The class that provides functionality of the role is `PFRole`. This class inherits from `PFObj`, so all the methods are accessible in the `PFRole` as well.

PFRole

The `PFRole` class provides you with the following additional properties other than the properties in `PFObj`:

- `name`: This field will store the name of the role, such as Administrators or Standard Users.
- `users`: This field will store the `PFUser` object as a relation. The user will inherit all the permissions of the role to which it's associated.
- `roles`: This field will hold the actual permissions for the `PFRole` object.

Securing the role object

The `PFRole` class uses the same ACL security like all the other Parse objects.

The following code will illustrate a way to create a new `PFRole` object:

```
// By specifying no write privileges for the ACL,
// we can ensure the role cannot be altered.
PFACL *roleACL = [PFACL ACL];
[roleACL setPublicReadAccess:YES];
PFRole *role = [PFRole
roleWithName:@"Administrator" acl:roleACL];
[role saveInBackground];
```

The previous code will create a new `role` object, which now can be associated with the `User` object. The user access control will be updated according to the `role` object.

The following code will associate `User` with the `role` object and update the user ACL with the new ACL provided by the role:

```
PFRole *role = [PFRole roleWithName:roleName
acl:roleACL];
for (PFUser *user in usersToAddToRole) {
    [role.users addObject:user];
}
for (PFRole *childRole in rolesToAddToRole) {
    [role.roles addObject:childRole];
}
[role saveInBackground];
```

Take great care while assigning ACLs to your roles, as they will directly impact the user's accessibility.

Securing other objects

Now you have learned to provide roles to the user. In the `PFACL` class of the user object, you can provide which users and roles should be granted read or write access to which object.

The following code will illustrate the usage of roles in `PFObject`:

```
PFObject *wallPost = [PFObject
objectWithClassName:@"WallPost"];
PFACL *postACL = [PFACL ACL];
[postACL setWriteAccess:YES
forRole:@"Administrator"];
```

```
wallPost.ACL = postACL;  
[wallPost saveInBackground];
```

Implementing role hierarchy

You can provide one role as the subset of another. This means that one role can contain another. You can create a parent-child relationship between roles. The result of such a relationship is that any permission granted to the parent will be implicitly granted to all its child roles.

For example, you can say that the `Administrator` role contains all the permissions that are granted in the `Moderator` role. The following code will explain the implementation of the role hierarchy:

```
PFRole *administrators = /* Your "Administrators"  
role */;  
PFRole *moderators = /* Your "Moderators" role */;  
[moderators.roles addObject:administrators];  
[moderators saveInBackground];
```

The previous code will add a relationship between the `Administrator` and `Moderator` role. Here, the `Administrator` role is a child role of `Moderator`.

Summary

In this chapter, we explored the users and role management of the application using Parse.

We started by exploring `PFUser` with its properties and login process.

Then, we learned about the security of users and other objects.

Finally, we explored `PFRole` to extend the user permissions and security along with grouping the users.

In the next chapter, we will learn about social media (Facebook, Twitter) integration in application using Parse.

Chapter 7. Social Media Integration

These days, social media integration is a common feature that all mobile applications incorporate. Some applications allow users to share data and updates to their social profiles such as Facebook and Twitter.

Nowadays, the traditional login process for the application using an e-mail ID and password is turning out to be cumbersome for the users. So to target such issues, many applications use Facebook and Twitter as their base login. Users can sign up using their Facebook or Twitter account. This even simplifies the steps of the login process and provides one-click sign up and login to the user. Keeping such things in mind, Parse provides you with the Plug and Play approach to integrate with social media, both for sharing and for the login process. In this chapter, you will learn in detail about social media integration using Parse.

Integrating Facebook

It's easy to integrate Facebook with your Parse application. The Facebook SDK can be used with the Parse SDK. You can link your `PFUser` with the Facebook user identities. You can use Facebook for sign up processes as well. Parse allows you to authenticate the user from Facebook and associate the user in your `PFUser` class.

In this section you will learn the integration of Facebook from your Parse application.

Prerequisites

To get started with the integration of Facebook with Parse, you need to set up the following things:

- Register on the Facebook developer portal (<https://developers.facebook.com/>) and create a Facebook application (<https://developers.facebook.com/docs/ios/getting-started/>), if you have not created it already.
- Facebook provides step-by-step procedure to create an app (<https://developers.facebook.com/apps>) and link it with the Facebook SDK.

Setting up the application

To get started with Facebook, you need to configure the following in your application:

1. Select your project file from the left-hand side panel, click on **Targets** and navigate to the **Info** tab.
2. Under the **Custom iOS Target Properties** section, add a new key with **Facebook App Id** as a string value and provide your Facebook app ID as the value.
3. Now add another key name with **URL Type**. Expand the key and at last you will see the **URL identifier**. Replace the **URL identifier** with **URL Schemes** and fill it with `fbYour_App_Id` (for example, Fb1234567890).

After updating your application with the Facebook keys, perform the following steps for setting up the code in your application:

1. In your `application:didFinishLaunchingWithOptions:` method, add the following line of code to initialize Facebook, just below the code where you have initialized the Parse SDK:

```
[PFFacebookUtils initializeFacebook];
```

2. Add the following methods in your `AppDelegate` file to handle the single sign-on feature of the Facebook SDK:

```

- (BOOL)application:(UIApplication
*)application openURL:(NSURL *)url
    sourceApplication:(NSString
*)sourceApplication annotation:(id)annotation
{
    return [PFFacebookUtils
handleOpenURL:url];
}

- (BOOL)application:(UIApplication
*)application handleOpenURL:(NSURL *)url {
    return [PFFacebookUtils
handleOpenURL:url];
}

```

Signing up on Facebook

Using the Parse SDK you can allow users to sign up using Facebook. The user object will also be created and saved for you by Parse. The following method will allow you to log in with Facebook:

```

[PFFacebookUtils loginWithPermissions:permissions
block:^(PFUser *user, NSError *error) {
    if (!user) {
        NSLog(@"Uh oh. The user cancelled the
Facebook login.");
    } else if (user.isNew) {
        NSLog(@"User signed up and logged in
through Facebook!");
    } else {
        NSLog(@"User logged in through
Facebook!");
    }
}];

```

In the previous code, `permissions` is an array of strings that specify what data your application wants to access from Facebook.

The following code will demonstrate the way you log in or sign up using Facebook:

```

- (IBAction)loginButtonTouchHandler:(id)sender {
    // The permissions requested from the user
    NSArray *permissionsArray = @[
@"user_about_me", @"user_relationships",
@"user_birthday", @"user_location"];

    // Login PFUser using Facebook
    [PFFacebookUtils
loginWithPermissions:permissionsArray
block:^(PFUser *user, NSError *error) {
    [_activityIndicator stopAnimating]; //
    Hide loading indicator

    if (!user) {
        if (!error) {
            NSLog(@"Uh oh. The user cancelled
the Facebook login.");
        } else {
            NSLog(@"Uh oh. An error occurred:
%@", error);
        }
    } else if (user.isNew) {
        NSLog(@"User with facebook signed up
and logged in!");
        [self.navigationController
pushViewController:[[UserDetailsViewController
alloc] initWithStyle:UITableViewStyleGrouped]
animated:YES];
    } else {
        NSLog(@"User with facebook logged
in!");
        [self.navigationController
pushViewController:[[UserDetailsViewController
alloc] initWithStyle:UITableViewStyleGrouped]
animated:YES];
    }
}];
}

```

On the button action you can add the previous code to log in or sign up using Facebook. In the previous code, we have used `permissionsArray`. This array will hold the strings to indicate the user data needed from Facebook.

After executing the preceding function, the following steps will take place:

1. The app will be redirected to Facebook or will prompt for permissions to access the data from Facebook.
2. When the user authenticates via Facebook, your application will receive a callback using the `handleOpenUrl` method that we had configured in the previous section.
3. The Parse SDK will fetch the data and save it to the `PFUser` class and will save the data on Parse if the user is new.
4. Your code block will be called with the user.

Parse provides you with a very deep integration with Facebook, as Parse has been acquired by Facebook, Inc. This also spells for even greater and deeper integration with Parse in future.

Publishing on Facebook

To post some updates on the user wall, you need to acquire publishing permissions from the user. The following code will acquire the publishing permission from the users:

```
[PFFacebookUtils reauthorizeUser:[PFUser
currentUser]

withPermissions:@[@"publish_stream"]

audience:PF_FBSessionDefaultAudienceFriends
              block:^(BOOL succeeded,
NSError *error) {
    if (succeeded) {
        //Your app now has publishing permissions
        for the user
    }
}];
```

Requesting user data

After user login, you can start fetching the user data from Facebook using the Graph API. The following code will demonstrate the way to

fetch the user data from Facebook:

```
- (void)fetchUserData {
    // ...
    // Create request for user's Facebook data
    FBRequest *request = [FBRequest requestForMe];

    // Send request to Facebook
    [request
startWithCompletionHandler:^(FBRequestConnection
*connection, id result, NSError *error) {
        if (!error) {
            // result is a dictionary with the
user's Facebook data
            NSDictionary *userData = (NSDictionary
*)result;

            NSString *facebookID =
userData[@"id"];
            NSString *name = userData[@"name"];
            NSString *location =
userData[@"location"][@"name"];
            NSString *gender =
userData[@"gender"];
            NSString *birthday =
userData[@"birthday"];
            NSString *relationship =
userData[@"relationship_status"];

            NSURL *pictureURL = [NSURL
URLWithString:[NSString
stringWithFormat:@"https://graph.facebook.com/%@/p
icture?type=large&return_ssl_resources=1",
facebookID]];

            // Now add the data to the UI elements
            // ...
        }
    }];
}
```

The `result` dictionary will hold all the requested data. You can fetch the data by querying with the key.

Linking users

In case of anonymous login or sign up using an e-mail ID and password, you can associate the existing `PFUser` to a Facebook account. The following code will illustrate a way to link an existing user to Facebook:

```
if (![PFFacebookUtils isLinkedWithUser:user]) {
    [PFFacebookUtils linkUser:user permissions:nil
    block:^(BOOL succeeded, NSError *error) {
        if (succeeded) {
            NSLog(@"User logged in with
Facebook!");
        }
    }];
}
```

After successful log in the `PFUser` class will be updated with the Facebook information.

The following code will unlink a Facebook account from a current user:

```
[PFFacebookUtils unlinkUserInBackground:user
block:^(BOOL succeeded, NSError *error) {
    if (succeeded) {
        NSLog(@"The user is no longer associated
with their Facebook account.");
    }
}];
```

Logging out

To log out the current user from the application, you simply need to call the `logout` method on the `PFUser` class. The following code will log out the user from the application:

```
- (void)logoutButtonAction:(id)sender {
    [PFUser logout]; // Log out
    // Return to login page
}
```

Keeping users logged in

It's cumbersome for the user to log in every time while launching the application. This is because Facebook session information is not stored in between launches. Parse makes this easy for us by internally saving the session in the `currentUser` object of the `PFUser` class.

In the launch view controller, you can check for the current session. If it exists, you can bypass the log in. The following code will illustrate how to bypass the log in:

```
- (void)viewDidLoad {
    ...
    if ([PFUser currentUser] && // Check if a user
        is cached
        [PFFacebookUtils isLinkedWithUser:[PFUser
        currentUser]]) // Check if user is linked to
        Facebook
    {
        // Push the next view controller without
        animation
        [self.navigationController
        pushViewController:

        [[UserDetailsViewController alloc]
        initWithStyle:UITableViewStyleGrouped]
        animated:NO];
    }
}
```

Integrating Twitter

Just like Facebook integration, Parse provides an easy way to integrate Twitter authentication as well. Just after a few lines of code, you will be able to log in with Twitter and save the data over the Parse Cloud.

Setting up Twitter

You need to perform the following steps to set up Twitter authentication using Parse:

1. If you have not created your application on Twitter already, you need to create your application on the Twitter developer portal (<https://dev.twitter.com/apps>).
2. Add your Twitter application's **Consumer Key** on the Parse application's settings page.
3. Enter a valid **Callback URL** for the Twitter application. This value is important for you to have control over this callback URL, so that the information isn't hijacked along the way. So, in this way, this value is used for authentication purposes by Twitter.
4. Add the following code in your `application:didFinishLaunchingWithOptions:` method, just below the code where you initialized Parse:

```
[PFTwitterUtils  
initializeWithConsumerKey:@"YOUR CONSUMER KEY"  
  
consumerSecret:@"YOUR CONSUMER SECRET"];
```

This is all you need to do to set up Twitter in your application. Now, we can allow the user to log in using Twitter.

Signing up and logging in

Parse provides you with the `PFTwitterUtils` class to allow your users to log in using Twitter. The following code will allow the user to sign up using Twitter:

```
[PFTwitterUtils loginWithBlock:^(PFUser *user,
NSError *error) {
    if (!user) {
        NSLog(@"Uh oh. The user cancelled the
Twitter login.");
        return;
    } else if (user.isNew) {
        NSLog(@"User signed up and logged in with
Twitter!");
    } else {
        NSLog(@"User logged in with Twitter!");
    }
}];
```

On executing the previous code, the following should happen:

- The user will be prompted for the permission to access Twitter authentication through an alert box or login dialog box
- After authorizing the application, the application will receive a callback
- Parse will receive the data on successful log in and will save it to the `PFUser` class on the cloud

Linking users

In case of anonymous login or sign up using an e-mail ID and password, you can associate the existing `PFUser` to a Twitter account. The following code will illustrate a way to link an existing user to Twitter:

```
if (![PFTwitterUtils isLinkedWithUser:user]) {
    [PFTwitterUtils linkUser:user block:^(BOOL
succeeded, NSError *error) {
        if ([PFTwitterUtils
isLinkedWithUser:user]) {
            NSLog(@"User logged in with
Twitter!");
        }
    }];
}
```

```
    }  
    }];  
}
```

After successful log in the `PFUser` class will be updated with the Twitter information.

The following code will unlink the Twitter account from a current user:

```
[PFTwitterUtils unlinkUserInBackground:user  
block:^(BOOL succeeded, NSError *error) {  
    if (!error && succeeded) {  
        NSLog(@"The user is no longer associated  
with their Twitter account.");  
    }  
}];
```

Summary

In this chapter, we explored the integration of the Facebook and Twitter SDK with the Parse SDK.

We started by exploring Facebook integration with setting up an application on the Facebook developer portal and then saw the login setup.

Then, we learned about publishing posts on Facebook, and linking users from Facebook.

Next, we learned about integrating in the application.

Finally, we explored Twitter integration and log in in our application.

In the next chapter, we will learn about the cloud functions and its usage in our application.

Chapter 8. Using Cloud Functions

In application development, all the processes cannot be executed on the client side. So it's recommended to execute such processes on the cloud end. Parse allows users to develop a mobile app, bypassing the server-side coding and management. While developing complex applications, users want some business logic to not be executed on the client end. So for such applications, Parse provides the Parse Cloud, where you can deploy your custom logic, which can be accessed by your application.

The cloud code is required to be written in JavaScript language. The difference would be the place of execution; cloud code will be executed on the Parse Cloud rather than executing on the mobile device. Once the cloud code is updated, it's available for use across all mobile environments instantly. Such features help you to change application behavior instantly with ease. In this chapter, we will learn about the implementation of the cloud code and its usage on the mobile end.

The cloud code

Before getting started with the cloud code, you need to set up the Parse command line tool. This tool will help you to manage and deploy your code on the cloud.

Installing the command-line tool

To install the Parse command-line tool on a Mac/Linux environment, you need to execute the following command in your terminal window:

```
curl -s
https://www.parse.com/downloads/cloud_code/install
er.sh | sudo /bin/bash
```

The previous line of code will install a tool named `parse` to the `/usr/local/bin/parse` directory. To uninstall, you just need to delete the file, as it does not have any junk files installed with it.

Setting up the cloud code

To set up the cloud code, create a directory where you want to save your cloud code. It's recommended to keep the cloud code in your project workspace.

The command `parse new` creates a new directory for you, and prompts for the selection of the application for which you are creating the cloud code:

```
$ parse new MyCloudCode
Email: demo@gmail.com
Password:
1:DemoApp
Select an App: 1
$ cd MyCloudCode
```

You need to use the e-mail ID and password of the Parse account to log in. In case of the `OAuth` login, you need to set a new password from your Parse settings to set up the cloud code. After successful execution of the previous command, the following file structure will be created for you:

```
-config/
  global.json
-cloud/
  main.js
-public/
  index.html
```

The explanation of the files mentioned in the preceding section is as follows:

- The JSON file in the `config` directory should not be edited, it's for Parse use
- The `cloud` directory is the place where you will store your cloud code
- Initially the `config` folder will contain `main.js`, which holds your cloud functions
- The `public` directory will hold the static data content for hosting on Parse

Creating the cloud function

To create the function on the cloud, you need to add the function to your `cloud/main.js` file. Let's take an example of the `cloud` function:

```
Parse.Cloud.define("demoCloudCode",
function(request, response) {
    response.success("Cloud integration is easy!");
});
```

To deploy the code on the Parse Cloud, run the following command in your terminal:

```
$ parse deploy
```

Once the function is deployed on the cloud, you can check the cloud code in your Parse application under the **Cloud Code** section. The deployed code should be published there. Now it's time to execute the cloud code on the mobile side. Parse provides you with the `callFunctionInBackground:withParameters:block:` method to execute the cloud code from the mobile end.

You will have to use the following code to execute the cloud code:

```
[PFCloud callFunctionInBackground:@"demoCloudCode"
        withParameters:@{}
```

```

                                block:^(NSString
*result, NSError *error) {
    if (!error) {
        NSLog(@"%@", result);
        // result is @"Cloud integration is easy!"
    }
}];

```

So, in this section, we explored how to write and execute the cloud code.

Hosting a website

It's easy to host a website on Parse. All the files in your `public` directory will be hosted at `your-custom-subdomain.parseapp.com`. The following code will illustrate a way to host your site on the Parse Cloud. Open the terminal and execute the following code:

```

$ echo "Hello World" > public/index.html
$ parse deploy

```

The previous line of code will add `Hello World` in your `index.html` file and with the `deploy` command your code will be deployed to the Parse Cloud.

To update your subdomain, navigate to the **Web Hosting** section of your app's setting. In the field of **ParseApp name**, provide a unique name, and all your website files will be available at `your-custom-subdomain.parseapp.com`.

Understanding complex functions

The cloud code is important to reduce the computation on the client side. Let's say, your application requires the average marks of students associated with the MBA course. The `Marks` object will look as follows:

```
{
  "student": "John Melon",
  "marks": 5,
  "course": "MBA"
}
```

In this case, you should not fetch a long list of data and filter that on the client side. You can add a Parse code for filtering the result and providing you with the resultant data. The following Parse Cloud code will help you to filter the data:

```
Parse.Cloud.define("averageMarks",
function(request, response) {
  var query = new Parse.Query("Marks");
  query.equalTo("course", request.params.course);
  query.find({
    success: function(results) {
      var sum = 0;
      for (var i = 0; i < results.length; ++i) {
        sum += results[i].get("marks");
      }
      response.success(sum / results.length);
    },
    error: function() {
      response.error("course lookup failed");
    }
  });
});
```

These are the following key points to note about the previous [Cloud](#) function:

- Define a [cloud](#) function, and a query on the [Marks](#) object
- Filter the [Marks](#) objects based on the [course](#) name provided in `params query.equalTo("course", request.params.course);`
- Then fetch the filtered data and return the average after completing the computation

Now to fetch the data using the cloud code on the client side, you need to use the following code:

```

[PFCloud callFunctionInBackground:@"averageMarks"
        withParameters:@[@"course":
@"MBA"]}
        block:^(NSNumber
*marks, NSError *error) {
    if (!error) {
        // marks is 45
    }
}];

```

The previous code will invoke the `cloud` function with the name `averageMarks`, and with the `course` name `MBA` as a parameter. After successful execution of the cloud code, you will get `marks` as the response, which will hold the average marks of the students who are associated with the MBA course.

Using cloud on the saved code

In case of multiplatform mobile application development, the cloud code saves a lot of code to write on the client end. Suppose you have an iOS, Android, or Windows environment for application development, then in that case you can avoid coding validation on all environments by deploying a validation code on the cloud. The Cloud provides you with the `beforeSave` method. The following code will illustrate the usage:

```

Parse.Cloud.beforeSave("averageMarks",
function(request, response) {
    if (request.object.get("marks") < 1) {
        response.error("you cannot give less than one
mark");
    } else if (request.object.get("marks") > 50) {
        response.error("you cannot give more than five
marks");
    } else {
        response.success();
    }
});

```

By this method, Parse allows you to add your custom logic, which will be executed before saving the object on the Cloud. In this

function you can add your validation code on the objects.

Likewise, Parse provides you with the `afterSave` method as well, which will help you to provide custom logic, which will be executed after saving the object:

```
Parse.Cloud.afterSave("averageMarks",
function(request) {
    query = new Parse.Query("Marks");
    query.get(request.object.get("course").course, {
        success: function(course) {
            post.increment("marks");
            post.save();
        },
        error: function(error) {
            console.error("Got an error " + error.code +
" : " + error.message);
        }
    });
});
```

The previous code will be executed after saving the object on the cloud. This method allows you to add custom logic, which will get executed after saving the object.

Using cloud on the deleted code

Just like saving, you can add the custom code, which will be executed before and after deleting any object from Parse. Parse provides you with the `beforeDelete` and `afterDelete` methods, which will help you to add your custom logic on these events.

The following code will illustrate the usage of the `beforeDelete` method. Similarly, you can use the `afterDelete` method as well:

```
Parse.Cloud.beforeDelete("Marks",
function(request) {
    query = new Parse.Query("Marks");
    query.equalTo("course", request.object.course);
    query.count({
        success: function(count) {
            if (count > 0) {
```

```
        response.error("Can't delete.");
    } else {
        response.success();
    }
},
error: function(error) {
    response.error("Error " + error.code + " : "
+ error.message + " when getting photo count.");
}
});
});
```

Summary

In this chapter, we explored the Parse Cloud code and its various implementations to simplify the application development on the mobile end.

We started by exploring the installation process of the command-line tool and then we saw the setup of the cloud code on Parse.

We also learned about writing the cloud functions and their usage on the client side.

Then, we learned about publishing our website on the cloud.

Finally, we explored complex-case scenarios and implementations using the cloud.

In the next chapter, we will learn about error handling and security on Parse.

Chapter 9. Error Handling and Security

While dealing with the cloud code, it's possible that a user could send invalid parameters in his/her queries. To handle such conditions on the cloud code, we have to send an error code in response to such queries. Errors can be in several forms, such as timeouts, ACL lists, undefined operations, and many more.

In this chapter, we will cover:

- Error handling in order to enhance your application stability to handle such errors and respond accordingly
- The measures to enhance the security of the data
- The way to export your Parse data

Error handling

While working with Parse, you may come across two types of errors. The first is the logical error, which can occur because of the way you are using the SDK. This kind of error will usually raise an exception called `NSException`.

Let's say in the case of logging in, the ideal login process is to provide a username and password before calling `signUp` on the `PFUser` class. So, according to the following code, you are bypassing the username and password and calling `signUp` directly:

```
PFUser *user = [PFUser user];  
[user signUp];
```

In such cases, the operation will throw an `NSInternalInconsistencyException` because `signUp` was called without username and password properties.

Errors can also occur while interacting with the Parse Cloud over the network. The error can be either because of connection with the Parse Cloud or it can be because of any requested operation. Let's look at another example:

```
- (void)fetchCourse {
    PFQuery *query = [PFQuery
queryWithClassName:@"Course"];
    [query
getObjectInBackgroundWithId:@"invalidObjectId"
                             target:self

selector:@selector(callbackForFetch:error:)];
}
```

In the previous code, we are trying to fetch the object which is not available on the Parse Cloud. In such cases, Parse will throw an error in the dictionary format where the error code is readable through `error.code`, and the message is readable through `error.userInfo`.

The following code demonstrates the `callback` method for the previous method call:

```
- (void)callbackForFetch:(PFObject *)result error:
(NSError *)error {
    if (result) {
        NSLog(@"Success!");
    } else {
        if ([error code] ==
kPFErrorObjectNotFound) {
            NSLog(@"Object not available!");
        } else if (error) {
            NSLog(@"Error: %@", [[error userInfo]
objectForKey:@"error"]);
        }
    }
}
```

In the preceding code, you will notice that we are using `kPFErrorObjectNotFound` for checking the error code. Parse provides

various predefined constants for the errors. You can find all the predefined error codes in the `PFConstants` header file.

There may be several other possible reasons for the query failure, such as the device being unable to connect to the Parse Cloud server. In such cases, you will receive an error message in response with the description of the error. For this the `callback` method will be something like the following piece of code:

```
- (void)callbackForFetch:(PFObject *)result error:
(NSError *)error {
    if (result) {
        NSLog(@"Success!");
    } else {
        if ([error code] ==
kPFErrorObjectNotFound) {
            NSLog(@"Object not available!");
            // Now also check for connection errors:
        } else if ([error code] ==
kPFErrorConnectionFailed) {
            NSLog(@"Connection failed with Parse
Cloud!");
        } else if (error) {
            NSLog(@"Error: %@", [[error userInfo]
objectForKey:@"error"]);
        }
    }
}
```

When you have sent `NSNumber` in response to the cloud code, its Boolean value is used to check whether the operation was successful or not. For example, you have invoked a `cloud` function to fetch the average marks of a student. In this case the `callback` method should be similar to the following method:

```
- (void)callbackForAverageMarks:(NSNumber *)result
error:(NSError *)error {
    if ([result boolValue]) {
        NSLog(@"Success!");
    } else {
        if ([error code] ==
kPFErrorConnectionFailed) {
```

```

        NSLog(@"Connection failed with
Cloud!");
    } else if (error) {
        NSLog(@"Error: %@", [[error userInfo]
objectForKey:@"error"]);
    }
}
}

```

In the previous method, if the Boolean value for the result is `true`, the operation succeeds, else it fails with an error. Error handling is the same for all the foreground tasks as well.

The request timeout for all the commotions is 10 seconds, so the synchronous calls will not hang for long period of time. The connection timeout is the same for synchronous and asynchronous tasks.

Parse provides you with the list of Error codes along with their explanations. You can check this list at <https://www.parse.com/docs/ios/api/Classes/PFConstants.html>.

Securing your application data

It is considered best practice in the software industry to secure your application data as much as possible. For securing your data, you can specify an access control list provided by Parse to provide security to your object based on the user. It's recommended to enable an anonymous user automatically when the application is launched. It's also recommended to provide public writability only on the required objects. Such settings will protect your application data from unauthorized access. Following code will illustrate the usage of security in your application:

```
[PFUser enableAutomaticUser];
PFACL *defaultACL = [PFACL ACL];
// Optionally enable public read access while
// disabling public write access.
// [defaultACL setPublicReadAccess:YES];
[PFACL setDefaultACL:defaultACL
withAccessForCurrentUser:YES];
```

It's recommended to provide ACL to all your objects, as you have already learned in [Chapter 6](#), *Users and Roles*.

You can secure your application even more by customizing the following settings on your **App Settings** page:

- Disable all the login mechanisms if you aren't using them. For example, if your app allows the user to log in using Facebook only, then disable all the other login methods from the settings.
- Provide your Facebook **App Id** and Twitter **Consumer Key** information in your Parse application settings page to enable server-side validation for user login attempts.

Exporting data

Parse also allows you to export your database. For exporting your data, navigate to the **Settings** page of the application and click on the **Export Data** button. Once you click on the **Export Data** button, Parse will compress all your data and export it to your e-mail ID, which is used for login. Exported data is in JSON format. The ZIP file will contain all your class objects in a separate file. The following is the example data of the `Class` object exported from Parse:

```
{ "results": [
  {
    "name": "buy groceries",
    "createdAt": "2013-07-03T11:04:05.062Z",
    "updatedAt": "2013-07-05T10:15:08.909Z",
    "objectId": "fYQ5ZfsM9m"
  },
  {
    "name": "team planning",
    "createdAt": "2013-08-15T07:22:51.976Z",
    "updatedAt": "2013-08-15T07:22:51.976Z",
    "objectId": "sBsAfr2Fm5"
  }
] }
```

Summary

In this chapter, we explored the Parse Cloud code and its various implementations to simplify the application development on the mobile end.

We started by exploring the types of errors and the ways to handle them in your project.

Then, we learned how to enhance the application security.

Finally, we have seen the way to export our data from Parse.

Index

A

- Access Control List / [Setting up Parse](#)
- accounts
 - setting up, on Parse / [Setting up accounts and applications on Parse](#)
- ACLWithUser method / [Securing objects](#)
- advanced push notifications
 - about / [Using advanced targeting](#)
 - pushes, sending to queries / [Sending pushes to queries](#)
 - customizing / [Customizing notifications](#)
 - platform-based targeting / [Platform-based targeting](#)
- anonymous user
 - creating / [Creating an anonymous user](#)
- App ID
 - setting up / [Setting up your App ID](#)
- Apple developer portal
 - URL / [Setting up your App ID](#)
- application data
 - securing / [Securing your application data](#)
- array-based queries
 - about / [Array-based queries](#)

B

- backend service, Parse
 - about / [Parse backend service](#)

C

- caching policies, queries
 - KPFCachePolicyIgnoreCache / [Caching queries](#)
 - KPFCachePolicyCacheOnly / [Caching queries](#)
 - KPFCachePolicyNetworkOnly / [Caching queries](#)
 - KPFCachePolicyCacheElseNetwork / [Caching queries](#)
 - KPFCachePolicyNetworkElseCache / [Caching queries](#)
 - KPFCachePolicyCacheThenNetwork / [Caching queries](#)
- channels
 - creating / [Channels](#)
 - subscribing to / [Subscribing to channels](#)
 - push notifications, sending to / [Sending push notifications to channels](#)
- Cloud code
 - command-line tool, installing / [Installing the command-line tool](#)
 - setting up / [Setting up the cloud code](#)
 - cloud function, creating / [Creating the cloud function](#)
 - website, hosting / [Hosting a website](#)
 - complex functions / [Understanding complex functions](#)
 - cloud, using on saved code / [Using cloud on the saved code](#)
 - cloud, using on delete code / [Using cloud on the deleted code](#)

- Cloud Code
 - about / [Cloud Code](#)
 - push notifications, sending through / [Cloud Code](#)
- command-line tool
 - installing / [Installing the command-line tool](#)
- compound queries
 - about / [Compound queries](#)

D

- data
 - exporting / [Exporting data](#)
- data types
 - about / [Data types](#)
- data usage, Parse analytics
 - about / [Data usage](#)
- delete method / [Updating the user object](#)

E

- error handling
 - about / [Error handling](#)
 - NSError / [Error handling](#)

F

- Facebook developer portal
 - URL / [Prerequisites](#)
- Facebook integration
 - about / [Integrating Facebook](#)
 - prerequisites / [Prerequisites](#)
 - application, setting up / [Setting up the application](#)
 - Facebook, signing up / [Signing up on Facebook](#)
 - Facebook, publishing on / [Publishing on Facebook](#)
 - user data, requesting / [Requesting user data](#)
 - users, linking / [Linking users](#)
 - logging out / [Logging out](#)
 - login, bypassing / [Keeping users logged in](#)
- files
 - handling / [Handling files](#)
 - PFFile / [PFFile](#)
 - image, saving / [Saving images](#)



- image
 - saving / [Saving images](#)
- iOS project
 - Parse, integrating in / [Integrating Parse in an iOS project](#)



- login process
 - signing up / [Signing up](#)

- user, verifying / [Verifying the user](#)
- logging in / [Logging in](#)
- user data, saving / [Saving the user's data](#)
- anonymous user, creating / [Creating an anonymous user](#)
- password, resetting / [Resetting passwords](#)
- user queries, writing / [Writing user queries](#)
- security, providing / [Providing security](#)
- logOut method / [Saving the user's data](#)

M

- methods
 - customizing / [Customizing properties and methods](#)

O

- object class method / [Initializing subclasses](#)

P

- Parse
 - relations / [Relations](#)
 - data types / [Data types](#)
 - queries / [Working with queries](#)
 - error handling / [Error handling](#)
 - application data, securing / [Securing your application data](#)
 - data, exporting / [Exporting data](#)
- Parse
 - backend service / [Parse backend service](#)
 - advantages / [Advantages of using Parse](#)
 - accounts, setting up / [Setting up accounts and applications on Parse](#)

- integrating, in iOS project / [Integrating Parse in an iOS project](#)
- setting up / [Setting up Parse](#)
- advanced push notifications / [Using advanced targeting](#)
- Parse analytics
 - integrating / [Integrating Parse analytics](#)
 - pages, tracking / [Tracking pages](#)
 - badges, clearing / [Clearing all badges](#)
 - data analytics, reading / [Reading analytics data](#)
 - data usage / [Data usage](#)
 - advantages / [Advantages of analytics](#)
- Parse App
 - push notifications, sending through / [Parse apps](#)
- Parse object
 - about / [Parse objects](#)
 - PFObject / [PFObject](#)
 - operations / [Operations on Parse objects](#)
 - saving / [Saving objects](#)
 - fetching / [Fetching objects](#)
 - saving offline / [Saving objects offline](#)
 - updating / [Updating objects](#)
 - counters, updating / [Updating counters](#)
 - data, storing in array format / [Storing data in an array format](#)
 - deleting / [Deleting objects](#)
- Parse website
 - push notifications, sending through / [Parse websites](#)
- password
 - resetting / [Resetting passwords](#)

- PFFile
 - about / [PFFile](#)
- PFInstallation
 - badges / [Installation](#)
 - channels / [Installation](#)
 - timeZone / [Installation](#)
 - deviceType / [Installation](#)
 - installationId / [Installation](#)
 - deviceToken / [Installation](#)
 - channelUri / [Installation](#)
 - appName / [Installation](#)
 - appVersion / [Installation](#)
 - parseVersion / [Installation](#)
 - appId / [Installation](#)
- PFObject
 - about / [PFObject](#), [Subclassing PFObject](#)
 - subclassing / [Subclassing PFObject](#)
- PFQuery class / [Creating queries](#)
- PFRole class
 - about / [PFRole](#)
 - properties / [PFRole](#)
 - role object, securing / [Securing the role object](#)
 - objects, securing / [Securing other objects](#)
 - role hierarchy, implementing / [Implementing role hierarchy](#)
- PFUser
 - about / [PFUser](#)
 - login process / [User login](#)

- predicate features types, not supported by Parse / [Using predicates](#)
- predicate features types, supported by Parse / [Using predicates](#)
- progress
 - tracking / [Tracking progress](#)
- properties
 - customizing / [Customizing properties and methods](#)
- provisioning profile
 - creating / [Creating your provisioning profile](#)
 - Parse App, configuring / [Configuring your Parse app](#)
 - iOS application, configuring for push notifications / [Configuring an iOS application for push notifications](#)
 - installing / [Installation](#)
- push notifications
 - setting up / [Setting up push notifications](#)
 - SSL certificate, creating / [Creating an SSL certificate](#)
 - certificate request, generating / [Generating certificate requests](#)
 - App ID, setting up / [Setting up your App ID](#)
 - sending / [Sending push notifications](#)
 - sending, through Parse website / [Parse websites](#)
 - sending, through Parse App / [Parse apps](#)
 - sending, through Cloud Code / [Cloud Code](#)
 - sending, Parse used / [Sending notifications](#)
 - channels / [Channels](#)
 - receiving / [Receiving push notifications](#)
 - payload, responding to / [Responding to payload](#)

Q

- queries
 - about / [Working with queries](#)
 - working with / [Working with queries](#)
 - predicates, using / [Using predicates](#)
 - query constraints, adding / [Adding query constraints](#)
 - relational queries / [Relational and complex queries](#), [Relational queries](#)
 - complex queries / [Relational and complex queries](#)
 - array-based queries / [Array-based queries](#)
 - string-based queries / [String-based queries](#)
 - compound queries / [Compound queries](#)
 - caching / [Caching queries](#)
 - creating / [Creating queries](#)
- query constraints / [Adding query constraints](#)

R

- relational queries
 - about / [Relational queries](#)
- relations
 - about / [Relations](#)
- requestPasswordResetForEmailInBackground* method / [Resetting passwords](#)
- role hierarchy
 - implementing / [Implementing role hierarchy](#)
- role object
 - securing / [Securing the role object](#)
- roles

- assigning / [Assigning roles](#)

S

- save method / [Updating the user object](#)
- security, login proces
 - providing / [Providing security](#)
 - user Object, updating / [Updating the user object](#)
 - objects, securing / [Securing objects](#)
- signUpInBackgroundWithBlock* method / [Signing up](#)
- signUp methods / [Signing up](#)
- social media integration
 - Facebook integration / [Integrating Facebook](#)
 - Twitter integration / [Integrating Twitter](#)
- SSL certificate
 - creating / [Creating an SSL certificate](#)
 - request, generating / [Generating certificate requests](#)
- string-based queries
 - about / [String-based queries](#)
- subclasses
 - initializing / [Initializing subclasses](#)

T

- Twitter developer portal
 - URL / [Setting up Twitter](#)

- Twitter integration
 - about / [Integrating Twitter](#)
 - Twitter, setting up / [Setting up Twitter](#)
 - signing up / [Signing up and logging in](#)
 - login / [Signing up and logging in](#)
 - user, linking / [Linking users](#)