Chapter 5

Integrating Third-Party Fitness Trackers and Data Using the Fitbit API

# Gheorghe Chesler

One would be remiss to discuss health sensors without mentioning the most popular connected motion tracker on the market, the Fitbit. Through its web-based API (application programming interface), Fitbit allows developers to access activity logged from its hardware, as well as related health information from Fitbit’s ecosystem, including meals and weight. This chapter will teach developers how to connect to the API from within their apps, as well as how to retrieve information from it and log new activities.

# Introduction to the Fitbit API

The Fitbit devices are health metrics trackers that record detailed minute-by-minute steps, distance, calorie burn data, and sleep records. The devices are popular because they are very light and cover the basic needs for health tracking.

The Fitbit tracker can record for at least seven days between sync opportunities. Beyond that, daily totals for step data, calorie counts, sleep records, and heart rate (if measured by your tracker) will be stored for 30 days. This data will all upload to your account as soon as you are able to sync and will then be reflected on your dashboard.

You can sync your tracker through any computer, as long as it has the Fitbit software installed and a base station (for Ultra tracker) or a wireless sync dongle (for all other trackers) plugged into the computer. There is, of course, a free iPhone app available that makes it possible to sync the data when you are on the road. You can only have one tracker paired to one Fitbit.com account.

The communication between the Fitbit tracker and the sync device is using a proprietary protocol that syncs the data with your online FitBit account. Fitbit provides an API that allows you to access your stored health data from the app you develop in Swift.

The Fitbit API is a RESTful API that provides access to FitBit data such as tracker collections, profile, and statistical data. This API is under continuous development and new features will be made available on an ongoing basis. The Fitbit API uses OAuth for authentication. The documentation for the Fitbit API can be found at https://dev.fitbit.com/docs

The Fitbit API allows you to interact with your account data found on the Fitbit server. It is important to realize that you will not be able to interact with and get the data from your device directly. If you do not have an Internet connection, the Fitbit device will store data in the Fitbit application, but that data will not reach the Fitbit server until you get back online, so you will only be able to get the latest version of your data when you have a stable Internet connection.

Recently Fitbit consolidated the API responses from XML to JSON (JavaScript Object Notation). In this chapter we will use requests that return data in JSON format. The API does not currently enforce SSL (Secure Sockets Layer); it is, however, recommended to use SSL for all communications, or at least for the OAuth handshake.

## The RESTful API

A **RESTful API** is built following a **RE**presentational **S**tate **T**ransfer architecture that defines best-practice rules for building scalable web services. Avoiding the complexity of SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language)-based APIs, a RESTful API usually relies on the HTTP verbs GET, PUT, POST, and DELETE to retrieve and send data to the remote servers. This data can be in a variety of formats, where JSON is the most popular.

**GET** is used to read data from a service, either as a set or as a unique record. Take, for example, the following request:

This will retrieve the user record with the id=123. It is up to the API to decide in what format to deliver the data, but in our case the data will be returned as JSON, something like:

{ “id” : “123”, “login”: “jsmith”, “firstName” : “Jim”, “lastName” : “Smith”}

Usually the GET request can be followed by a URL (uniform resource locator)-encoded string of parameters, as key-value pairs, as in the following example:

GET https://api.genericapi.com/v1/item?color=green&size=large

We see this all the time in URLs, so there is nothing unusual about it. Another method is to encode the params as part of the URL.

GET https://api.genericapi.com/v1/item/color/green/size/large

The documentation of the API you implement should give full detail on how to compose a GET request such as

GET https://api.genericapi.com/v1/user

**PUT** is used to update a known record. Per the specification, PUT replaces the known record data with a set of different values. Depending on who is implementing the API, PUT can be also used to update just a subset of values instead of changing the entire data record. If it deviates from the specification, we usually find a clear definition of what PUT does in the documentation of the API.

When an id is defined in the request, PUT will replace the addressed entity, and if it does not exist, it will create it. PUT must contain a payload, which is the data record it updates. Look, for example, at the following requests:

PUT https://api.genericapi.com/v1/user/123

{ “login”: “jsmith”, “firstName” : “Jane”, “lastName” : “Smith”}

PUT https://api.genericapi.com/v1/user

{ “id” : “123”, “login”: “jsmith”, “firstName” : “Jane”, “lastName” : “Smith”}

The two requests should have the same effect, updating the user with the id=123. Depending on the implementation, the API will return either the updated record or just a 200 OK response.

The PUT operation is known as idempotent, meaning that no matter how many times you repeat it given the same data, the result will be the same.

**POST** is used to create a new record. It is uncommon for the POST action to take an element id, as the API usually assigns the id. It is common for the POST action to return the inserted record, populated with the assigned record id. Take, for example, the following request/response:

POST https://api.genericapi.com/v1/user

{ “login”: “jdoe”, “firstName” : “John”, “lastName” : “Doe”}

Response(200 OK):

{ “id” : “133”, “login”: “jdoe”, “firstName” : “John”, “lastName” : “Doe”}

**DELETE** is used to delete records, given their record id. It usually returns no response other than the regular 200 OK responses. There is not much to it other than the URL containing the id to be deleted.

DELETE https://api.genericapi.com/v1/user/123

### The return format

Usually the return format for a request is specified in the header of the request, with the key-value pair.

Accept: application/json

Some APIs choose instead to use a file extension appended to the URL of the request, to recognize the desired format for the response. In this case there is no requirement for the return format specified in the header. This is the case for the Fitbit API, as we will see further on. Taking examples from above, following is how it would look:

GET <https://api.genericapi.com/v1/item/color/green/size/large.json>

POST https://api.genericapi.com/v1/user.json

A RESTful API takes the REST specification as a guideline, and you will rarely see this implemented strictly. It is not uncommon to see POST being used instead of PUT, or the other way around, or even using a POST to delete an item.

## Fitbit RESTful API implementation details

Taking the general ideas defined in the previous section, here are some things that are particular to the Fitbit API:

* The patterns used in the service URL
* The format of POST data

The URL for a service is segmented in a service and object/subservice, with some magic added to it. Normally, when we GET from a service, the preferred order is the API version, followed by the service name, followed by the id of the entity, then a GET string, URL-encoded, and separated from the base URL by a question mark. In the case of PUT/POST operations, the agreed-upon standard is to send the data in the body of the message.

Fitbit composes the service URL a bit differently, making everything a part of the URL, and avoiding dealing with more complexity and eventually performance impact on its end. Take, for example, the following calls:

GET /1/user/228TQ4/profile.json

GET /1/user/-/profile.json

The user id is 228TQ4 but is followed by the name of the object/subservice being requested, and the extension shows the preferred response format, so there is no need to set the HTTP header field Accept. To make matters more interesting, if the user id is the id of the current user, we need to specify a dash (this is the magic part). If we consider the fact that the id is the user id, not the profile id, we see here the assumption being made that a user can have only one profile, and there is no need to expose the profile id.

Here is an example where the GET parameters are encoded in the URL path. Following are the resource URLs for getting body weight data:

GET /<api-version>/user/-/body/log/weight/date/<date>.<response-format>

GET /<api-version>/user/-/body/log/weight/date/<base-date>/<period>.<response-format>

GET /<api-version>/user/-/body/log/weight/date/<base-date>/<end-date>.<response-format>

Here are actual calls following the foregoing patterns:

GET /1/user/-/body/log/weight/date/2010-02-21.json

GET /1/user/-/body/log/weight/date/2010-03-27/1w.json

GET /1/user/-/body/log/weight/date/2010-03-27/2010-04-15.json

Somebody at Fitbit must have thought that this was a good idea, but there is a free-style assumption made here. Normally, even if you decide to use the URL as a kitchen sink, it would be a good idea to keep things consistent: instead, we look at “body/log/weight/date” to represent something like “these are the keys for the values that follow,” but even that does not hold, since the date can stand for one date (start), begin-end dates, or start date followed by the period.

Composing a URL for a request should not require a degree in creative writing, but that’s the Fitbit API and we need to adapt to it, so great care has to be taken when writing services that make a specific request, given that there is no clear rule of composition for the URLs of Fitbit’s API requests.

For POST requests, things are again rather different than the standard usage. The Fitbit API requires us to send the POST data as a URL-encoded string, as part of the POST URL. One of the advantages of sending POST data in the body of the message is that it would not land in the HTTP server activity logs, and it would not be exposed in the actual URL, but then it would be a bit more tedious to sign a request. This is perhaps the reason that Fitbit decided to take this approach, as much as it is not the most secure. Examples of POST requests follow:

POST /1/user/-/bp.json?date=2015-04-24&weight=73

POST /1/user/-/bp.json?date=2015-04-23&diastolic=80&systolic=120

From the current implementation we can see that Fitbit has limited the available verbs to just GET and POST, not making active use of PUT or DELETE. This can, of course, change with the ongoing development of the API.

## Setting up a local playground with Apache

To be able to test our code way before we are ready to deal with the complexities of the OAuth implementation, we need to set up a local web server. The safe assumption is that you are working on a Mac, so Apache is already installed for you.

For OSX versions before Yosemite, you can use Web Sharing to set up a local web server. OS X 10.10 Yosemite comes with the Apache 2.4 pre-installed, but there is no longer a Web Sharing preference pane in System Preferences.

You can install one of the online available Web Sharing preference panes or simply use the provided apachectl command from your terminal.

To start up the web server, simply bring up the Terminal (/Applications/Utilities/Terminal) and type the following:

$ sudo apachectl start

To stop apache you would type:

$ sudo apachectl stop

If you start Apache now, you can reach your server in a browser by pointing it at the URL: http://127.0.0.1/ or even http://localhost/, and you should see a simple header that says: “**It works!**”

The document that contains the "It works!" text is called index.html.en and is located in the /Library/WebServer/Documents folder. Your current user does not own this folder, so you will not have permissions right away to create files. The easiest way to get permissions there is to change the flags on that folder. For that, you have to execute a command as root, using sudo.

$ sudo chmod 777 /Library/WebServer/Documents

You will be asked to enter your user password, which is the same password you use to log in to your Mac. By default, the users on a Mac have administrator privileges, so you should be able to use sudo to change flags or ownership of files and folders that are not owned by your user.

Now your user can create new documents in that folder: this is where we will create the two test documents used to verify the requests made by the first version of the APIClient library.

A more elaborate way would be to edit the Apache config file, and point it to a folder in your home folder. There is enough information available online on how to do this: for the time being, all we need is to create two test files in that folder, and start Apache, so we will take the easy route.

### Creating the test documents

The easiest way is to use the same Terminal screen and vi to create the files. If you are not familiar with using vi, you can use any other popular console text editor (nano, cat, or echo):

vi /Library/WebServer/Documents/data.json

Paste the following text into vi, then save the file:

{"Response":{"key":"value"}}

To edit the second file

vi /Library/WebServer/Documents/badData.json

paste the following text into vi, then save the file.

{"Response”:{{“key":"value"}}

The second file is intentionally populated with malformed JSON: this will allow us to test the handling of bad or incomplete responses that we might get from the API. Remember, just because you are talking to a public API does not mean it will always work perfectly, or return valid responses: your code will have to be able to compensate and handle the errors properly.

If you already started Apache, there is no need to restart it: what we added are two static documents and Apache will properly recognize them when they are present in the document folder.

## The OAuth1.0a authentication model

The OAuth1.0a authentication model is a rather complex set of interactions between the consumer and the service provider, which is in our case the Fitbit API.

In a simplified way, the following steps are required to access your protected resources:

1. The consumer requests a request token
2. Using the request token, the consumer obtains the user authorization
3. The consumer then requests an access token from the service provider
4. Using the access token, the consumer can now make requests to access the protected resources

To use the Fitbit API we need to sign up for a developer account and register the application. Once we do so, we get the following bits of information that we will use in crafting the requests to the Fitbit API:

* Client (consumer) key
* Client (consumer) secret
* Temporary credentials (request token) URL
* Token credentials (access token) URL
* Authorize URL

The authentication information is usually passed in the header of the request, as key-value pairs. This allows you to build test cases or run tests from the command line, as in the example in Listing 5-1.

Listing 5-1.

$ curl -X POST -i -H 'Authorization: OAuth oauth\_consumer\_key="abcd1234", oauth\_nonce="123", oauth\_signature="q4567aacc%3D", oauth\_signature\_method="HMAC-SHA1", oauth\_timestamp="1429137772", oauth\_version="1.0"' https://api.fitbit.com/oauth/request\_token

One important note here is that the curl example request is in one line. Curl is a command-line tool that allows us to perform a GET/POST request and retrieve the contents of the request.

## The Fitbit OAuth implementation

The Fitbit API is making a transition from OAuth 1.0a to OAuth 2.0 as authentication protocol. In this chapter we will use the OAuth 1.0a, which is the current production version. When Fitbit will transition to OAuth 2.0, you will have to change your application to use the new version.

The upgrade from one authentication protocol to another is usually staged, so that first the new protocol is made available to developers, then as a beta to the general public. The developers migrate their code to the new protocol at their convenience, and that equates to using a different base URL for each authentication protocol, so both authentication protocols will be available for a reasonable period of time.

When the API operators observe from the amount of API traffic that the old authentication protocol is not being used, or has minimal usage, only then will they declare the old protocol obsolete and announce a date of when the support for it will be discontinued.

As a developer it is good to keep in touch with the latest documentation for the APIs you are implementing, so that you can give yourself ample time to upgrade your application to the new authentication protocol.

For the current OAuth 1.0a implementation, Fitbit follows strictly the specification of the protocol.

Following are the steps to making an OAuth 1.0a request:

1. The client acquires a key and secret from Fitbit by registering an application at dev.fitbit.com.
2. The client builds an application that uses content from Fitbit.
3. The user requests to view content in the client application.
4. The client requests and receives temporary credentials from Fitbit.
5. The client redirects the user to Fitbit in order for the user to authorize the client application.
6. The user approves the client application, and Fitbit redirects the user to the client application site, passing a verifier.
7. The client requests and receives token credentials from Fitbit using the verifier it received.
8. Using token credentials, the client makes calls to access Fitbit resources on behalf of the user.

To help with the implementation of the current authentication protocol, the Fitbit developer site has an OAuth tutorial page under “Authentication.”

**Step 5** involves a bit more work in the UI (user interface), such as displaying the redirect URL in a browser. On that page you will have a form where you will have to agree that the app can have read/write access to your account. Upon confirmation of access, you will get a verifier code, which you will need to use to request the token credentials used in this chapter.

**The code in this chapter will work with the token credentials obtained in Step 8**: we will not do a complete implementation; we will be focusing instead on creating the basics for signing OAuth requests and making the API requests once the token credentials are obtained.

**You need to complete Steps 3 to 7 on your own.** The signing mechanism for the requests is the same—just the list of elements to be used in the signature is different, but very well documented on the Fitbit support site, where you can double-check your process by testing every step by hand and comparing the output with the output of your code.

Once we established the signature process for the common-case scenario (Step 8) it will be very easy for you to create functions that sign with a different set of parameters, as we will show later on.

## Fitbit API call rate limits

The documentation for API call rate limits can be found on the Fitbit page under "Basics/Rate Limits"/.

At the time of writing of this book, the Fitbit API set limits to the number of calls that can be made to the API in a given amount of time. When doing the full implementation in your application, your code has to be aware and able to handle the case when it hits one of the rate limits shown in the sections “The Client+Viewer Rate Limit” and “The Client Rate Limit.”

The Fitbit API has two separate rate limits on the number of calls you can make. Both are hourly limits that reset at the start of the hour.

### The Client+Viewer Rate Limit

All Read calls (as well as a couple of sensitive Read & Write) are rate limited with the current quota of 150 calls/hour. You can make 150 API requests per hour for each user who has authorized your application to access his or her data. This rate limit is applied when you make an API request using your application's consumer key and secret with the user's access token and token secret.

### The Client Rate Limit

Your application can make 150 API requests per hour without a user access token and token secret. These types of API requests are for retrieving non-user data, such as Fitbit's general resources.

### Response Headers

Fitbit API responses include headers that provide you with your rate limit status.

* Fitbit-Rate-Limit-Limit: the quota number of calls
* Fitbit-Rate-Limit-Remaining: the number of calls remaining before hitting the rate limit
* Fitbit-Rate-Limit-Reset: the number of seconds until the rate limit resets

### Hitting the Rate Limit

Your application will receive an HTTP 429 response from the Fitbit API when a request is not fulfilled due to reaching the rate limit. A "Retry-After" header is sent with the number of seconds until your rate limit is reset and you can begin making calls again.

## Making async calls

The Fitbit API is an external resource that might or might not be available to your device. This depends on your Internet connection availability, as well as any factors that prevent your device from accessing the Fitbit API. Additionally, sometimes API calls take longer than expected.

For this reason, the calls we make to the API need to be made as async calls. As an application user, this gives you the ability to do other things in your app, while the application is taking its time to communicate with the API and make the data transfers to and from the Fitbit API.

The simplest form of an async call is the following:

Listing 5-2.

var url: NSURL = NSURL(string: ”http://127.0.0.1/data.json")!

var request = NSMutableURLRequest(URL: url)

request.HTTPMethod = “GET”

NSURLConnection.sendAsynchronousRequest(request, queue: NSOperationQueue.mainQueue()) {

(urlResponse : NSURLResponse!, data : NSData!, error: NSError!) -> Void in

// do something here with the response data

}

The block of code after the list of parameters of sendAsynchronousRequest() is the code being called when the API sends back a response and the connection is closed. The data object is the content of the API response, as an NSData object. This is necessary since it is not a guarantee that the API will always return JSON: the service being called could deliver an image or other binary file, which we will have to handle as necessary. For the purpose of this chapter we will assume that our responses are always JSON strings, and our code will handle the response accordingly.

The other side of async calls is that we do not have direct control over their flow—once launched, we need to provide a handler for the response async call that was triggered as the result of a button click. Your code needs to disable that button action to prevent duplicate calls from happening until a valid response has been received and the data has been successfully processed.

## Using callbacks as parameters

We aggregate most of the API functionality in the APIClient.swift library. This makes sense, because we don’t want to duplicate this code for every type of call we make, and every service.

Listing 5-3.

Our apiRequest function has the following signature:

func apiRequest (

service: APIService,

method: APIMethod,

id: String!,

urlSuffix: NSArray!,

inputData: [String:String]!,

callback: (responseJson: NSDictionary!, responseError: NSError!) -> Void ) {

// Api call code here

}

We can see that one of the parameters is the callback parameter, which defines the signature of the callback function being passed to it. The callback function will be called in the block of code passed to the async call, after the API response has been received.

Following is the sample code for a generic GET handler using our APIClient:

Listing 5-4.

func getData (service: APIService,

id: String!=nil, urlSuffix: NSArray!=nil, params: [String:String]!=[:]) {

var blockSelf = self

var logger: UILogger = viewController.logger

self.apiRequest(

service,

method: APIMethod.GET,

id: id,

urlSuffix: urlSuffix,

inputData: params,

callback: { (responseJson: NSDictionary!, responseError: NSError!) -> Void in

if (responseError != nil) {

logger.logEvent(responseError!.description)

// Handle here the error response in some way

} else {

blockSelf.processGETData(service, id: id, urlSuffix: urlSuffix, params: params, responseJson: responseJson)

}

})

}

We see here that we make a copy of self in the blockSelf variable. This is necessary, since the callback code block is a closure that does not have the context of the caller. If we pass self directly, this will block it from being garbage collected when it goes out of scope or gets destroyed. In this case self is the APIClient instance we use to make API calls, and we will run into memory leaks over time if we create/destroy the APIClient instance.

# Setting up a Fitbit-compatible iOS project

To implement OAuth in Swift, we could use an available library that supports both OAuth 1.0a and OAuth 2.0 to simplify the transition when Fitbit will default to OAuth 2.0. For the purpose of this book, we will implement the OAuth layer ourselves, instead of relying on a third-party library. We will also show how to make use of some Objective-C libraries where needed, to avoid re-inventing the wheel in Swift, and keep the scope of the Swift code small. Creating

We begin by creating an empty, single-page project. This chapter aims to show how to communicate with the Fitbit API, not how to build an UI interface around it, so our application will be minimalistic, exposing just a few UI elements to trigger actions and track the communication with the Fitbit API, and making inline changes to some view controller functions as we go along.

## The view controller

A basic view controller for this chapter will only show a few buttons and a text area that we will use to display the communication with the API.

To initialize and to be able to use these buttons and fields, they have to be assigned macros that make them available/visible in the Interface Builder. We also define the variables used for the API and logger objects. Since these will be initialized at a later time, these variables need to be defined in the view controller as optional:

Listing 5-5.

class ViewController: UIViewController {

@IBOutlet var labelButton : UIButton!

@IBOutlet var textArea : UITextView!

var api: APIClient!

var logger: UILogger!

In the viewDidLoad() function we initialize the API object, as well as the log library that will output text to our textArea field. The content and functionality of these libraries will be explained as we go.

Listing 5-6.

override func viewDidLoad() {

super.viewDidLoad()

api = APIClient(parent: self)

logger = UILogger(out: textArea)

}

To assign an action to a button, we create a function that performs the action, and is also annotated with the proper macro to make it available in the Interface Builder. We will add a log statement to show the beginning of the request, and we can also change the title of the button, while it is pressed.

Listing 5-7.

@IBAction func clickButton() {

logger.logEvent(“=== Good Request ===")

api.getData(APIService.GOOD\_JSON)

labelButton.setTitle("Good Request Sent", forState: UIControlState.Normal)

}

@IBAction func unclickButton() {

labelButton.setTitle("=== Good Request ===", forState: UIControlState.Normal)

}

Notice that we used the APIService.GOOD\_JSON and APIService.BAD\_JSON service names. This is a first-step implementation that uses mock services from a local server and gets in return a pre-formatted, static content, so that we can test the code. This will be later replaced by actual services like, for example, APIService.ACCOUNT or APIService.PROFILE. These are distinct values from enums that we define in the APIClient.swift library.

We can wire these button actions in the storyboard (Figure 5-1).

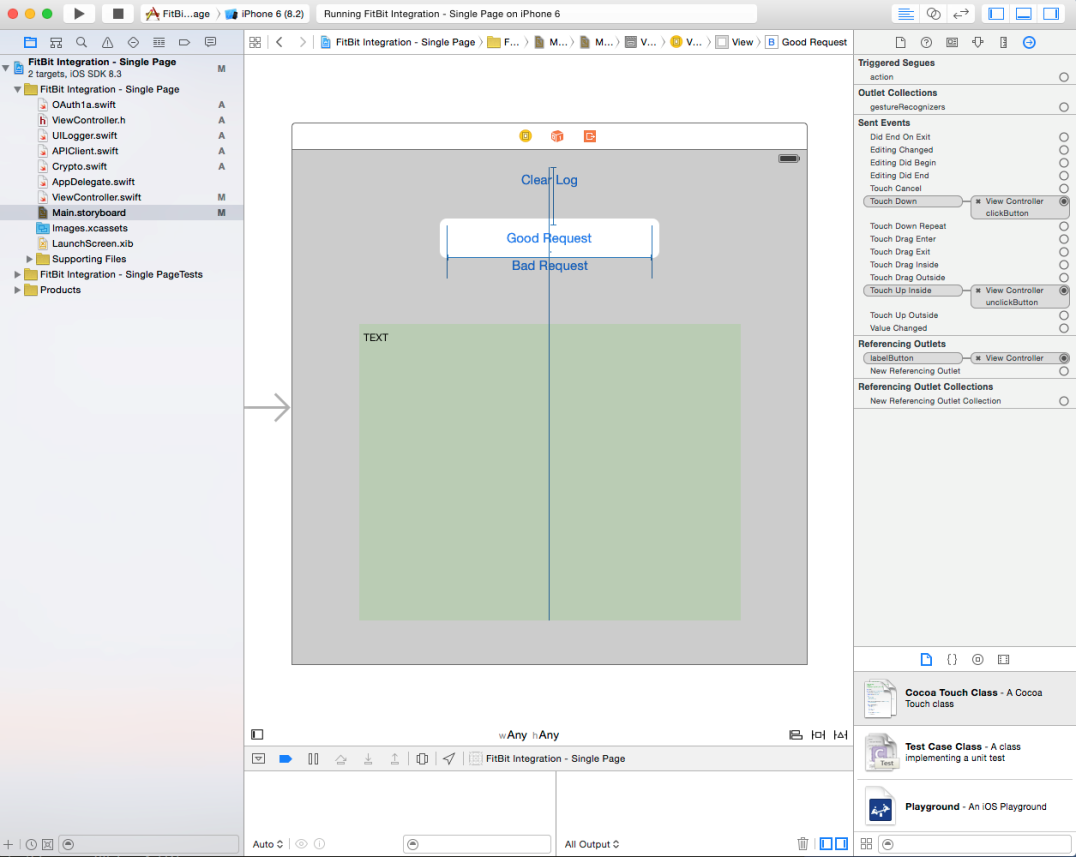


Figure 5-1. Wiring actions in the storyboard

This is the entire ViewController.swift code that we will use to test the internal wiring of the APIClient library. You might notice that we commented out the goLive() method, that would set the baseURL for the API to the liveBaseURL in the APIClient library(more on this later).

Listing 5-8.

import UIKit

class ViewController: UIViewController {

@IBOutlet var clearButton : UIButton!

@IBOutlet var labelButton : UIButton!

@IBOutlet var labelButton2 : UIButton!

@IBOutlet var textArea : UITextView!

var api: APIClient!

var logger: UILogger!

required init(coder aDecoder: NSCoder) {

super.init(coder: aDecoder)

}

override func viewDidLoad() {

super.viewDidLoad()

api = APIClient(parent: self)

logger = UILogger(out: textArea)

// api.goLive()

}

override func didReceiveMemoryWarning() {

super.didReceiveMemoryWarning()

// Dispose of any resources that can be recreated.

}

@IBAction func unclickButton() {

labelButton.setTitle("Good Request", forState: UIControlState.Normal)

}

@IBAction func unclickButton2() {

labelButton2.setTitle("Bad Request", forState: UIControlState.Normal)

}

@IBAction func clickButton() {

logger.logEvent("=== Good Request ===")

api.getData(APIService.GOOD\_JSON)

labelButton.setTitle("Good Request Sent", forState: UIControlState.Normal)

}

@IBAction func clickButton2() {

logger.logEvent("=== Bad Request ===")

api.getData(APIService.BAD\_JSON)

labelButton2.setTitle("Bad Request Sent", forState: UIControlState.Normal)

}

@IBAction func clickClearButton() {

logger.clear()

}

}

When you implement your application, you will most likely have a series of requests triggered by either a sync button or a timer, all the time keeping in mind that the API has a call rate limitation.

## The logger library

The logger library was assigned a variable in the view controller that will keep an instance of the logger around with the proper target assigned—in our case we use a text area field for the activity logging.

To keep things simple, we implement just a couple of functions that will allow us to track the API activity. These functions will interact with the textArea field we set up in the view controller. Just as in the view controller, the textArea field is declared optional, as it will be initialized in the init() function. The code in Listing 5-9 goes in the UILogger.swift file:

Listing 5-9.

import Foundation

import UIKit

class UILogger {

var textArea : UITextView!

required init(out: UITextView) {

dispatch\_async(dispatch\_get\_main\_queue()) {

self.textArea = out

};

self.clear()

}

func clear() {

dispatch\_async(dispatch\_get\_main\_queue()) {

self.textArea!.text = ""

}

}

func logEvent(message: String) {

dispatch\_async(dispatch\_get\_main\_queue()) {

self.textArea!.text = textArea!.text.stringByAppendingString("=> " + message + "\n")

}

}

}

## Setting up a basic set of crypto functions

Since the data to and from the API is of String type, the easiest way to set up the basic crypto functions is to set up a few extensions on the String object to handle SHA1 and HMAC hashing (the SHA and HMAC are hashing algorithms). We can set this in a separate file in our project, called Crypto.swift.

The sha1() function is provided as a convenience method that allows you to create fingerprints of results when testing against the API. It is not used in the OAuth signing process, and was implemented just to show how something like this could be done.

The hmac() function is used for creating the OAuth signature. You can verify its functionality with the Google OAuth request signature test page at http://oauth.googlecode.com/svn/code/javascript/example/signature.html. This is an easier-to-use test resource than the one provided by the Fitbit developer site, and it allows you to provide any combination of keys and tokens.

The escapeUrl() function is conveniently placed in the same context, because we will use it for composing the signature base string. It could have been a library function in its own right, but since it is conceivable that other parts of the product might use a good escaping tool, we decided to use it as an overload to the String object. Believe it or not, there is no NSCharacterSet with this set of characters, which is pretty odd, as this is the most useful set when composing an URL string.

There are a couple of enums in this library that are used by the hmac() function: we could have just used the values for SHA1 and saved ourselves the extra work with the enums, but with the full list configured, you can reuse this code for any other API that uses another hashing method than SHA1.

The functions are relying on Objective-C code that does the heavy work, so we need to set up a bridging header file. For that, create a new file in the project, give it a name(Crypto.h) and a location(Fitbit Integration - Single Page), and associate it with the current folder, as seen in Figure 5-2.

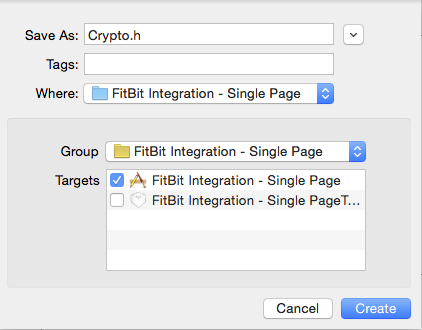


Figure 5-2. Saving the new crypto header file

Listing 5-10 shows the content of the Crypto.h file: we added two lines that include the code resources we will use for calculating SHA1 checksums and HMAC-SHA1 signatures. The ifndef section needs to stay as it is.

Listing 5-10.

#import <CommonCrypto/CommonCrypto.h>

#import <CommonCrypto/CommonHMAC.h>

#ifndef FitBit\_Integration\_\_\_Single\_Page\_Crypto\_h

#define FitBit\_Integration\_\_\_Single\_Page\_Crypto\_h

#endif

Once the header file has been created, we need to move it up in the project file hierarchy, by drag-and-dropping it to the top of the file list. Then we need to set up the path for the bridging header (Fitbit Integration - Single Page/Crypto.h) in our project under **Build Settings / Swift Compiler - Code Generation**, as shown in Figure 5-3.

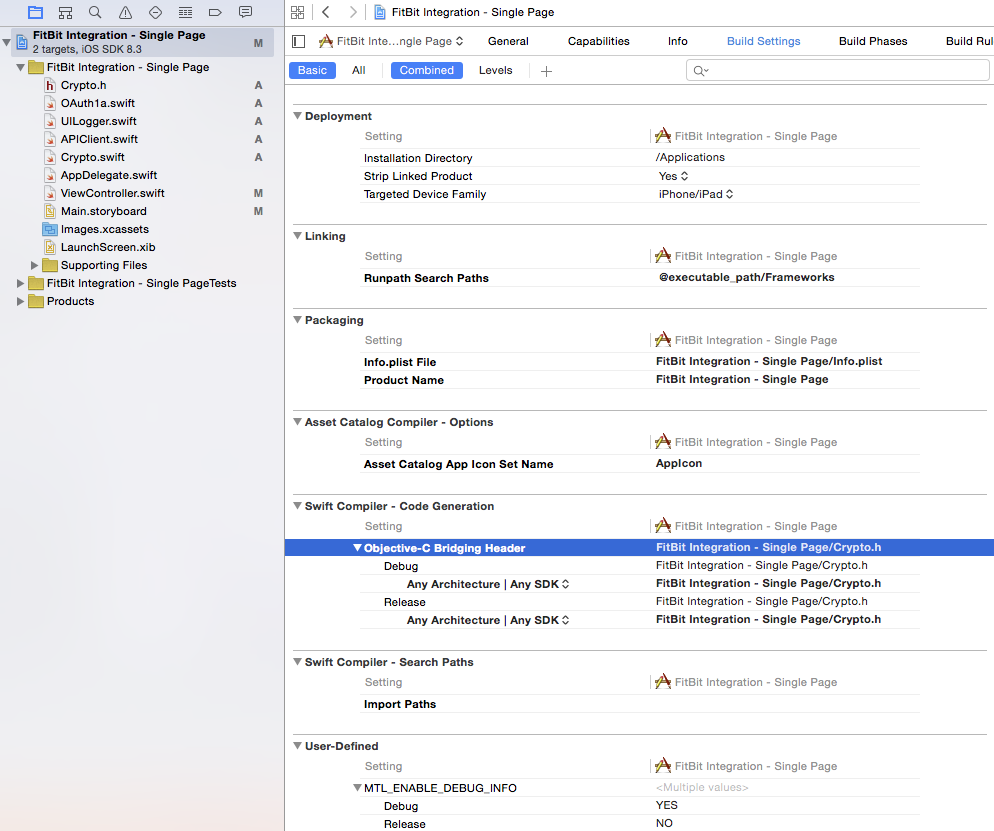


Figure 5-3. Setting up the path for the bridging header in the project

### The full content of the Crypto.swift

Listing 5-11.

import Foundation

extension String {  
 func sha1() -> String {  
 let data = self.dataUsingEncoding(NSUTF8StringEncoding)!  
 var digest = [UInt8](count:Int(CC\_SHA1\_DIGEST\_LENGTH), repeatedValue: 0)  
 CC\_SHA1(data.bytes, CC\_LONG(data.length), &digest)  
 let output = NSMutableString(capacity: Int(CC\_SHA1\_DIGEST\_LENGTH))  
 for byte in digest {  
 output.appendFormat("%02x", byte)  
 }  
 return output  
 }  
  
 func hmac(algorithm: HMACAlgorithm, key: String) -> String {  
 let str = self.cStringUsingEncoding(NSUTF8StringEncoding)  
 let strLen = Int(self.lengthOfBytesUsingEncoding(NSUTF8StringEncoding))  
 let digestLen = algorithm.digestLength()  
 let result = UnsafeMutablePointer<CUnsignedChar>.alloc(digestLen)  
 let objcKey = key as NSString  
 let keyStr = objcKey.cStringUsingEncoding(NSUTF8StringEncoding)  
 let keyLen = Int(objcKey.lengthOfBytesUsingEncoding(NSUTF8StringEncoding))  
 CCHmac(algorithm.toCCHmacAlgorithm(), keyStr, keyLen, str!, strLen, result)  
 let data = NSData(bytes: result, length: digestLen)  
 result.destroy()  
 return  
data.base64EncodedStringWithOptions(NSDataBase64EncodingOptions.Encoding64CharacterLineLength)  
 }  
   
 func escapeUrl() -> String {  
 var source: NSString = NSString(string: self)  
 var chars = "abcdefghijklmnopqrstuvwxyz"  
 var okChars = chars + chars.uppercaseString + "0123456789.~\_-"  
 var customAllowedSet = NSCharacterSet(charactersInString: okChars)  
 return source.stringByAddingPercentEncodingWithAllowedCharacters(customAllowedSet)!  
 }  
}  
enum HMACAlgorithm {  
 case MD5, SHA1, SHA224, SHA256, SHA384, SHA512  
 func toCCHmacAlgorithm() -> CCHmacAlgorithm {  
 var result: Int = 0  
 switch self {  
 case .MD5:  
 result = kCCHmacAlgMD5  
 case .SHA1:  
 result = kCCHmacAlgSHA1  
 case .SHA224:  
 result = kCCHmacAlgSHA224  
 case .SHA256:  
 result = kCCHmacAlgSHA256  
 case .SHA384:  
 result = kCCHmacAlgSHA384  
 case .SHA512:  
 result = kCCHmacAlgSHA512  
 }  
 return CCHmacAlgorithm(result)  
 }  
   
 func digestLength() -> Int {  
 var result: CInt = 0  
 switch self {  
 case .MD5:  
 result = CC\_MD5\_DIGEST\_LENGTH  
 case .SHA1:  
 result = CC\_SHA1\_DIGEST\_LENGTH  
 case .SHA224:  
 result = CC\_SHA224\_DIGEST\_LENGTH  
 case .SHA256:  
 result = CC\_SHA256\_DIGEST\_LENGTH  
 case .SHA384:  
 result = CC\_SHA384\_DIGEST\_LENGTH  
 case .SHA512:  
 result = CC\_SHA512\_DIGEST\_LENGTH } return Int(result)  
 }  
}

## The API client library

We created this library for the functions that make async requests to the API. You can find the complete code at the end of this section; for now we will discuss parts of the API client library code. The header of the class contains the URLs and other variables needed for the API functionality.

Listing 5-12.

class APIClient {

var apiVersion: String!

var baseURL: String = "http://127.0.0.1"

var liveBaseURL: String = "https://api.fitbit.com"

var liveAPIVersion: String = "1"

var requestTokenURL: String = "https://api.fitbit.com/oauth/request\_token"

var accessTokenURL: String = "https://api.fitbit.com/oauth/access\_token"

var authorizeURL: String = "https://www.fitbit.com/oauth/authorize"

var viewController: ViewController!

var oauthParams: NSDictionary!

var oauthHandler: OAuth1a!

required init (parent: ViewController!) {

viewController = parent

oauthParams = [

"oauth\_consumer\_key" : "6cf4162a72ac4a4382c098caec132782",

"oauth\_consumer\_secret" : "c652d5fb28f344679f3b6b12121465af",

"oauth\_token" : "5a3ca2edf91d7175cad30bc3533e3c8a",

"oauth\_token\_secret" : "da5bc974d697470a93ec59e9cfaee06d",

]

oauthHandler = OAuth1a(oauthParams: oauthParams)

}

We added the oauthParams in the init function for convenience—your code will have to gather/compose these values when you do a complete implementation of the OAuth/signup process. The requestTokenURL, accessTokenURL, and authorizeURL were also added to show where they would best be located, but they are not used in the code of this chapter.

When you write tests, you might want to change between the liveBaseURL and liveAPIVersion and a local test URL that is the default for the baseURL, as we do in our case to test the good and bad JSON. A simple function will allow you to switch to live mode, as follows:

func goLive () {

baseURL = liveBaseURL discussing

apiVersion = liveAPIVersion

}

A generic function to perform a GET from a service looks as shown in Listing 5-13.

Listing 5-13.

func getData (service: APIService, id: String!=nil, urlSuffix: NSArray!=nil, params: [String:String]!=[:]) {

var blockSelf = self

var logger: UILogger = viewController.logger

self.apiRequest(

service,

method: APIMethod.GET,

id: id,

urlSuffix: urlSuffix,

inputData: params,

callback: { (responseJson: NSDictionary!, responseError: NSError!) -> Void in

if (responseError != nil) {

logger.logEvent(responseError!.description)

// Handle here the error response in some way

}

else {

blockSelf.processGETData(service, id: id, urlSuffix: urlSuffix, params: params, responseJson: responseJson)

}

}

)

}

For the urlSuffix we use the NSArray data type that will hold all elements of the URL being accessed. We saw in The Fitbit OAuth implementation that there is no clear rule to composing an API URL; the API service calls have instead service names and values mashed up in a slash-delimited string. Since some of these could be numbers, the NSArray type is ideal because by default it contains AnyObject elements. We also pass the urlSuffix to the processGETData function, so that we can make a decision about what to do with the response, given the service being called, the optional id of the item, and the urlSuffix. We also defined default values for urlSuffix and params to allow our functions to make calls without providing all nil parameters in tow.

The optional input params is a dictionary with strings for keys and values. This is the most convenient format, considering that POST is not any different from GET in the way the parameters are passed to the API.

The block passed to the NSURLConnection.sendAsynchronousRequest is a closure, which is why we need to assign the blockSelf variable that will be used to make calls in the context of the APIClient library.

The function would be the actual handler of that response, which takes the generic form.

func processGETData (service: APIService, id: String!, urlSuffix: NSArray!, params: [String:String]!=[:], responseJson: NSDictionary!) {

// do something with data here

}

Just like the GET request, the POST request can have the same structure:

Listing 5-14.

func postData (service: APIService, id: String!=nil, urlSuffix: NSArray!=nil, params: [String:String]!=[:]) {

var blockSelf = self

var logger: UILogger = viewController.logger

self.apiRequest(

service,

method: APIMethod.POST,

id: id,

urlSuffix: urlSuffix,

inputData: params,

callback: { (responseJson: NSDictionary!, responseError: NSError!) -> Void in

if (responseError != nil) {

logger.logEvent(responseError!.description)

// Handle here the error response in some way

}

else {

blockSelf.processPOSTData(service, id: id, urlSuffix: urlSuffix, params: params, responseJson: responseJson)

}

})

}

func processPOSTData (service: APIService, id: String!, urlSuffix: NSArray!, params: [String:String]!=[:], responseJson: NSDictionary!) {

// do something with data here

}

Of course, we can implement the request process in many different ways, but having a common handler for an API request type allows us to avoid a callback hell.

We notice that the verb is not a string but an enum value: APIMethod.GET. This is an enum that we define in this library, to provide easy access to the verbs as strings, rather than using the strings directly. It also gives us control on which HTTP verbs are supported by the API client. This code goes at the end of APIClient.swift.

Listing 5-15.

enum APIMethod {

case GET, PUT, POST, DELETE

func toString() -> String {

var method: String!

switch self {

case .GET:

method = "GET"

case .PUT:

method = "PUT"

case .POST:

method = "POST"

case .DELETE:

method = "DELETE"

}

return method

}

func hasBody() -> Bool {

var hasBody: Bool

switch self {

case .GET:

hasBody = false

case .PUT:

hasBody = true

case .POST:

hasBody = true

case .DELETE:

hasBody = false

}

return hasBody

}

}

The hasBody() function is provided here as an example that could be useful in the apiRequest to properly format the request, so that GET and DELETE use the parameters as key-value pairs, while the PUT and POST use it as JSON. This is not necessary in our case, since the Fitbit API does not actually use a POST body but, rather, does a POST to a URL with the data formatted as URL-encoded parameters.

There is another enum we define in the APIClient library that provides shortcuts to actual services via the toString() function. We saw this in the ViewController used as APIService.GOOD\_JSON. We will extend this later to add other services and also provide a function to return the suffix we might want to use for some calls, but for now this is the basic format. As in Listing 5-15, the code in Listing 5-16 goes at the end of APIClient.swift.

Listing 5-16.

enum APIService {

case USER, ACTIVITIES, FOODS, GOOD\_JSON, BAD\_JSON

func toString() -> String {

var service: String!

switch self {

case .USER:

service = "user"

case .ACTIVITIES:

service = "activities"

case .FOODS:

service = "foods"

case .GOOD\_JSON:

service = "data"

case .BAD\_JSON:

service = "badData"

}

return service

}

}

We added the extra bits like GOOD\_JSON and BAD\_JSON to allow us to do internal testing. These bits point to the test pages we created when we set up the local Apache playground. Since our apiRequest function will handle adding the .son suffix to each URL, we only use the file base name.

Next to be defined is the apiRequest() function. This function will make the actual API request, and that includes handling the OAuth signing and eventual verification of the response data. The method signature is showing that the only required params are the service, the method, and the callback function.

Listing 5-17.

func apiRequest (

service: APIService,

method: APIMethod,

id: String!,

urlSuffix: NSArray!,

inputData: [String:String]!,

callback: (responseJson: NSDictionary!, responseError: NSError!) -> Void ) {

// Code goes here

}

The services currently available are USER, ACTIVITIES, FOODS—the API overloads them with a variable list of params, so in essence your calls will need to provide the larger APIService, and then provide via the urlSuffix the URL path extension to point to the right resource. This will be explained in more detail later on.

As for the content of the method, following is what we need to do for an API request:

* Compose the base URL of the service
* Add the URL suffix if it was specified
* Add the extension for the return data type
* Create the OAuth signature and populate the request headers
* Serialize and append to the URL the input params
* Make the API request as an async call

In the code block passed to the async call, we also need to do the following:

* Verify the OAuth signature of the response (if provided)
* De-serialize the JSON response
* Call the callback function

To compose the base URL of the service, we use the following code in Listing 5-18:

Listing 5-18.

var serviceURL = baseURL + "/"

if apiVersion != nil {

serviceURL += apiVersion + "/"

}

serviceURL += service.toString()

if id != nil && !id.isEmpty {

serviceURL += "/" + id

}

var request = NSMutableURLRequest()

request.HTTPMethod = method.toString()

If you recall, we saw in the Fitbit API implementation details that when an id is not provided, it is replaced by a dash in some requests. This kind of magic is not something we want to handle here: instead we rely on the caller to provide an id if one is needed, or the dash otherwise.

In the same segment, we create the request object and assign it the request method. The serviceURL is still being composed, so it would be premature to assign it to the request at this point.

If this API would support a JSON request body for POST requests, we could use something like the code in Listing 5-19 to serialize the input data.

Listing 5-19.

var error: NSError?

request.HTTPBody = NSJSONSerialization.dataWithJSONObject(inputData, options: nil, error: &error)

if error != nil {

callback(responseJson: nil, responseError: error)

return

}

request.addValue("application/json", forHTTPHeaderField: "Content-Type")

Unfortunately the Fitbit API is not that fancy, and it takes instead a simple URL-encoded set of params appended to the POST URL.

To handle the composition of the URL, we create the asURLString() function. This takes a dictionary of input params and creates a URL-encoded string, with the parameters sorted alphabetically. Sorting the parameters is not a requirement in the URL request, but we will use the same code in the OAuth library.

Listing 5-20.

func asURLString (inputData: [String:String]!=[:]) -> String {

var params: [String] = []

for (key, value) in inputData {

params.append( "=".join([ key.escapeUrl(), value.escapeUrl()] ))

}

params = params.sorted{ $0 < $1 }

return "&".join(params)

}

The URL suffix needs to be made part of the URL—we got in the input an NSArray of strings or numbers that will be used to compose the suffix—they will be all reduced to a simple string, appended to the base URL.

// The urlSuffix contains an array of strings that we use to compose the final URL

if urlSuffix?.count > 0 {

serviceURL += "/" + urlSuffix.componentsJoinedByString("/")

}

Adding the extension for the return data type is a rather simple matter. We also set the HTTP header for Accept, even if the Fitbit API does not require it. This is good practice, and it is possible that they will end up using it at some point.

// All URLs need to have have at least the .json suffix, if not already defined

if !serviceURL.hasSuffix(".json") && !serviceURL.hasSuffix(".xml") {

serviceURL += ".json"

}

request.addValue("application/json", forHTTPHeaderField: "Accept")

To create the OAuth signature and populate the request headers, we will make use of the crypto library for the hmac encoding. The OAuth1a library instance prepared as the oauthHandler is used to sign the request, given an optional list of parameters in urlParameters. The extra argument signUrl is not used here and is not shown because the method signature defines a default value (nil) for it, but it could be used when signing with a partial URL like in the case of getting the temporary token (not shown in this chapter).

Before we create the OAuth signature, we need to assign the following serviceURL to the request:

request.URL = NSURL(string: serviceURL)

oauthHandler.signRequest(request, urlParameters: urlParameters)

Now we are ready to make the API request as an async call. Note how we created a local variable logger that points to the logging handler of the view controller—this is necessary because inside the closure we don’t have visibility to variables and functions from the current library or from ViewController. The callback block for the async calls contains the basic code needed to handle the result data and call the callback function that we got when apiRequest() was invoked. Once again, when interpreting the response, an error can occur parsing the JSON data, which will be handled by the callback function.

To parse an API response into a JSON object, we use an NSDictionary object that will hold any combination of key-values. This is necessary since the API responses can contain any combination of numbers, strings, arrays, dictionaries, and NSDictionary supports by default AnyObject types. The NSJSONReadingOptions.MutableContainers specifies that arrays and dictionaries be created as mutable objects.

var jsonResult: NSDictionary?

var rData: String = NSString(data: data, encoding: NSUTF8StringEncoding)! as String

if data != nil {

jsonResult = NSJSONSerialization.JSONObjectWithData(data, options: NSJSONReadingOptions.MutableContainers, error: &error) as? NSDictionary

}

When encountering an error case that we need to report, we can create our own error object. To do this in Swift, we use the following approach:

error = NSError(domain: "response", code: -1, userInfo: ["reason":"blank response"])

We added some logging for the response data, with an example on how to pretty-print JSON to the text area used for logging. We do want to format the response in such a way that is easy to read, and pretty-printed JSON appears as one key-value per line, nicely indented.

Listing 5-21.

var blockSelf = self  
var logger: UILogger = viewController.logger  
NSURLConnection.sendAsynchronousRequest(request, queue: NSOperationQueue.mainQueue()) {  
 (urlResponse : NSURLResponse!, data : NSData!, error: NSError!) -> Void in  
 //the request returned with a response or possibly an error  
 logger.logEvent("URL: " + serviceURL)  
 var error: NSError?  
 var jsonResult: NSDictionary?  
 if urlResponse != nil {  
 blockSelf.extractRateLimits(urlResponse)  
 var rData: String = NSString(data: data, encoding: NSUTF8StringEncoding)! as String  
 if data != nil {  
 jsonResult = NSJSONSerialization.JSONObjectWithData(data, options: NSJSONReadingOptions.MutableContainers, error: &error) as? NSDictionary  
 }  
 var logResponse: String! = blockSelf.prettyJSON(jsonResult)  
 logResponse == nil  
 ? logger.logEvent("RESPONSE RAW: " + (rData.isEmpty ? "No Data" : rData) )  
 : logger.logEvent("RESPONSE JSON: \(logResponse)" )  
 print("RESPONSE RAW: \(rData)\nRESPONSE SHA1: \(rData.sha1())")  
 }  
 else {  
 error = NSError(domain: "response", code: -1, userInfo: ["reason":"blank response"])  
 }  
 callback(responseJson: jsonResult, responseError: error)  
}

Displaying pretty-formatted JSON can be useful in other places too, so we extracted the following code in the prettyJSON() function:

func prettyJSON (json: NSDictionary!) -> String! {  
 var pretty: String!  
 if json != nil && NSJSONSerialization.isValidJSONObject(json!) {  
 if let data = NSJSONSerialization.dataWithJSONObject(json!, options: NSJSONWritingOptions.PrettyPrinted, error: nil) {  
 pretty = NSString(data: data, encoding: NSUTF8StringEncoding) as? String  
 } } return pretty

}

To parse the response and extract the API rate limits, we need to extract from the response header the following key-value pairs:

Fitbit-Rate-Limit-Limit: 150

Fitbit-Rate-Limit-Remaining: 149

Fitbit-Rate-Limit-Reset: 1478

The function extractRateLimits() will take care of that, and it will also throw some statements in the console log that will help with the debugging. We already defined the variables in the APIClient header, and we update these with every API call we make. Since we have the rateLimitTimeStamp value, we can use this to compare with the current timestamp and see whether the rateLimitTimeStamp + rateLimitReset is smaller than the current timestamp; then we can make the next API call with confidence and otherwise handle the issue inside the application, returning an error early, instead of making a call that we know is going to fail. This can be easily implemented in the apiRequest() so we leave this as an exercise for the reader.

Listing 5-22.

func extractRateLimits (response: NSURLResponse) {

if let urlResponse = response as? NSHTTPURLResponse {

if let rl = urlResponse.allHeaderFields["Fitbit-Rate-Limit-Limit"] as? NSString as? String {

rateLimit = rl.toInt()

print("RESPONSE HEADER rateLimit: \(rl)")

}

if let rlr = urlResponse.allHeaderFields["Fitbit-Rate-Limit-Remaining"] as? NSString as? String {

rateLimitRemaining = rlr.toInt()

print("RESPONSE HEADER rateLimitRemaining: \(rlr)")

}

if let rlx = urlResponse.allHeaderFields["Fitbit-Rate-Limit-Reset"] as? NSString as? String {

rateLimitReset = rlx.toInt()

rateLimitTimeStamp = String(format:"%d", Int(NSDate().timeIntervalSince1970)).toInt()

print("RESPONSE HEADER rateLimitReset: \(rlx), checked at: \(rateLimitTimeStamp)")

}

}

}

### The code for APIClient.swift

Listing 5-23 shows the code we have so far for the APIClient library (APIClient.swift):

Listing 5-23.

import Foundation

class APIClient {

var apiVersion: String!

var baseURL: String = "http://127.0.0.1"

var liveBaseURL: String = "https://api.fitbit.com"

var liveAPIVersion: String = "1"

var requestTokenURL: String = "https://api.fitbit.com/oauth/request\_token"

var accessTokenURL: String = "https://api.fitbit.com/oauth/access\_token"

var authorizeURL: String = "https://www.fitbit.com/oauth/authorize"

var viewController: ViewController!

var oauthParams: NSDictionary!

var oauthHandler: OAuth1a!

var rateLimit: Int!

var rateLimitRemaining: Int!

var rateLimitReset: Int!

var rateLimitTimeStamp: Int!

required init (parent: ViewController!) {

viewController = parent

oauthParams = [

"oauth\_consumer\_key" : "6cf4162a72ac4a4382c098caec132782",

"oauth\_consumer\_secret" : "c652d5fb28f344679f3b6b12121465af",

"oauth\_token" : "5a3ca2edf91d7175cad30bc3533e3c8a",

"oauth\_token\_secret" : "da5bc974d697470a93ec59e9cfaee06d",

]

oauthHandler = OAuth1a(oauthParams: oauthParams)

}

func goLive () {

baseURL = liveBaseURL

apiVersion = liveAPIVersion

}

func postData (service: APIService, id: String!=nil, urlSuffix: NSArray!=nil, params: [String:String]!=[:]) {

var blockSelf = self

var logger: UILogger = viewController.logger

self.apiRequest(

service,

method: APIMethod.POST,

id: id,

urlSuffix: urlSuffix,

inputData: params,

callback: { (responseJson: NSDictionary!, responseError: NSError!) -> Void in

if (responseError != nil) {

logger.logEvent(responseError!.description)

// Handle here the error response in some way

}

else {

blockSelf.processPOSTData(service, id: id, urlSuffix: urlSuffix, params: params, responseJson: responseJson)

}

})

}

func processPOSTData (service: APIService, id: String!, urlSuffix: NSArray!, params: [String:String]!=[:], responseJson: NSDictionary!) {

// do something with data here

}

func getData (service: APIService, id: String!=nil, urlSuffix: NSArray!=nil, params: [String:String]!=[:]) {

var blockSelf = self

var logger: UILogger = viewController.logger

self.apiRequest(

service,

method: APIMethod.GET,

id: id,

urlSuffix: urlSuffix,

inputData: params,

callback: { (responseJson: NSDictionary!, responseError: NSError!) -> Void in

if (responseError != nil) {

logger.logEvent(responseError!.description)

// Handle here the error response in some way

}

else {

blockSelf.processGETData(service, id: id, urlSuffix: urlSuffix, params: params, responseJson: responseJson)

}

})

}

func processGETData (service: APIService, id: String!, urlSuffix: NSArray!, params: [String:String]!=[:], responseJson: NSDictionary!) {

// do something with data here

}

func apiRequest (

service: APIService,

method: APIMethod,

id: String!,

urlSuffix: NSArray!,

inputData: [String:String]!,

callback: (responseJson: NSDictionary!, responseError: NSError!) -> Void ) {

// Compose the base URL

var serviceURL = baseURL + "/"

if apiVersion != nil {

serviceURL += apiVersion + "/"

}

serviceURL += service.toString()

if id != nil && !id.isEmpty {

serviceURL += "/" + id

}

var request = NSMutableURLRequest()

request.HTTPMethod = method.toString()

// The urlSuffix contains an array of strings that we use to compose the final URL

if urlSuffix?.count > 0 {

serviceURL += "/" + urlSuffix.componentsJoinedByString("/")

}

// All URLs need to have have at least the .json suffix, if not already defined

if !serviceURL.hasSuffix(".json") && !serviceURL.hasSuffix(".xml") {

serviceURL += ".json"

}

request.addValue("application/json", forHTTPHeaderField: "Accept")

request.URL = NSURL(string: serviceURL)

// Sign the OAuth request here

oauthHandler.signRequest(request, urlParameters: inputData)

if !inputData.isEmpty {

serviceURL += "?" + asURLString(inputData: inputData)

request.URL = NSURL(string: serviceURL)

}

//now make the request

var blockSelf = self  
var logger: UILogger = viewController.logger  
NSURLConnection.sendAsynchronousRequest(request, queue: NSOperationQueue.mainQueue()) {  
 (urlResponse : NSURLResponse!, data : NSData!, error: NSError!) -> Void in  
 //the request returned with a response or possibly an error  
 logger.logEvent("URL: " + serviceURL)  
 var error: NSError?  
 var jsonResult: NSDictionary?  
 if urlResponse != nil {  
 blockSelf.extractRateLimits(urlResponse)  
 var rData: String = NSString(data: data, encoding: NSUTF8StringEncoding)! as String  
 if data != nil {  
 jsonResult = NSJSONSerialization.JSONObjectWithData(data, options: NSJSONReadingOptions.MutableContainers, error: &error) as? NSDictionary  
 }  
 var logResponse: String! = blockSelf.prettyJSON(jsonResult)  
 logResponse == nil  
 ? logger.logEvent("RESPONSE RAW: " + (rData.isEmpty ? "No Data" : rData) )  
 : logger.logEvent("RESPONSE JSON: \(logResponse)" )  
 print("RESPONSE RAW: \(rData)\nRESPONSE SHA1: \(rData.sha1())")  
 }  
 else {  
 error = NSError(domain: "response", code: -1, userInfo: ["reason":"blank response"])  
 }  
 callback(responseJson: jsonResult, responseError: error)  
}

func asURLString (inputData: [String:String]!=[:]) -> String {

var params: [String] = []

for (key, value) in inputData {

params.append( "=".join([ key.escapeUrl(), value.escapeUrl()] ))

}

params = params.sorted{ $0 < $1 }

return "&".join(params)

}

func prettyJSON (json: NSDictionary!) -> String! {

var pretty: String!

if json != nil && NSJSONSerialization.isValidJSONObject(json!) {

if let data = NSJSONSerialization.dataWithJSONObject(json!, options: NSJSONWritingOptions.PrettyPrinted, error: nil) {

pretty = NSString(data: data, encoding: NSUTF8StringEncoding) as? String

}

}

return pretty

}

func extractRateLimits (response: NSURLResponse) {

// Fitbit-Rate-Limit-Limit: 150

// Fitbit-Rate-Limit-Remaining: 149

// Fitbit-Rate-Limit-Reset: 1478

if let urlResponse = response as? NSHTTPURLResponse {

if let rl = urlResponse.allHeaderFields["Fitbit-Rate-Limit-Limit"] as? NSString as? String {

rateLimit = rl.toInt()

print("RESPONSE HEADER rateLimit: \(rl)")

}

if let rlr = urlResponse.allHeaderFields["Fitbit-Rate-Limit-Remaining"] as? NSString as? String {

rateLimitRemaining = rlr.toInt()

print("RESPONSE HEADER rateLimitRemaining: \(rlr)")

}

if let rlx = urlResponse.allHeaderFields["Fitbit-Rate-Limit-Reset"] as? NSString as? String {

rateLimitReset = rlx.toInt()

rateLimitTimeStamp = String(format:"%d", Int(NSDate().timeIntervalSince1970)).toInt()

print("RESPONSE HEADER rateLimitReset: \(rlx), checked at: \(rateLimitTimeStamp)")

}

}

}

}

enum APIService {

case USER, ACTIVITIES, FOODS, GOOD\_JSON, BAD\_JSON

func toString() -> String {

var service: String!

switch self {

case .USER:

service = "user"

case .ACTIVITIES:

service = "activities"

case .FOODS:

service = "foods"

case .GOOD\_JSON:

service = "data"

case .BAD\_JSON:

service = "badData"

}

return service

}

}

enum APIMethod {

case GET, PUT, POST, DELETE

func toString() -> String {

var method: String!

switch self {

case .GET:

method = "GET"

case .PUT:

method = "PUT"

case .POST:

method = "POST"

case .DELETE:

method = "DELETE"

}

return method

}

## }The OAuth library

This library (OAuth1a.swift) handles the signing of the requests, not making the request themselves. There is more work for you to do to integrate all the steps in the OAuth process, and the best choice here is to build a good foundation, get the signing process right, and extend this later as needed.

The header of the library contains the essential variables specific to the signing process like the signatureMethod, oauthVersion and all the keys and tokens involved in any request. The init function assigns values for these, if they were provided. Depending on the request we are signing, we might need just a subset of them.

This code in Listing 5-24 and all other code in this section is saved in the OAuth1a.swift file:

Listing 5-24.

import Foundation

class OAuth1a {

var signatureMethod: String = "HMAC-SHA1"

var oauthVersion: String = "1.0"

var oauthConsumerKey: String!

var oauthConsumerSecret: String!

var oauthToken: String!

var oauthTokenSecret: String!

required init (oauthParams: NSDictionary) {

oauthConsumerKey = oauthParams.objectForKey("oauth\_consumer\_key") as! String

oauthConsumerSecret = oauthParams.objectForKey("oauth\_consumer\_secret") as! String

oauthToken = oauthParams.objectForKey("oauth\_token") as! String

oauthTokenSecret = oauthParams.objectForKey("oauth\_token\_secret") as! String

}  
...  
}

We saw that the elements that help randomize the signature are the timestamp and the nonce. The timestamp is just the epoch time and can be easily read from NSDate. We are only interested in the Int value of this, which is the epoch time in seconds. This is found in the signRequest() function listed next.

let timeStamp = String(format:"%d", Int(NSDate().timeIntervalSince1970))

We created a separate function for the nonce, which makes things very easy for the caller. It takes an alphanumeric string that contains all valid characters then runs a random pointer over the string and extracts a character at that index until we get a random string in the desired length.

Listing 5-25.

func randomStringWithLength (len : Int) -> String {

let letters : NSString = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

var randomString : NSMutableString = NSMutableString(capacity: len)

for (var i=0; i < len; i++){

var length = UInt32(letters.length)

var rand = arc4random\_uniform(length)

randomString.appendFormat("%C", letters.characterAtIndex(Int(rand)))

}

return randomString as String

}

### Signing the request

The main function in this library is signRequest(). It takes the request object, the URL parameters, as well as an optional alternative URL to be used for signing the request. The first thing we do is to prepare the timeStamp, nonce, and the URL to be used for signing. These are the variable parts in our signature. To make debugging easy we added some console logging statements. There is no need to log these to the text area on the screen, since it would crowd the space and provide little value once we get the first request right.

Listing 5-26.

func signRequest (request: NSMutableURLRequest, urlParameters: [String:String]!=[:], signUrl: String!=nil) {

let timeStamp = String(format:"%d", Int(NSDate().timeIntervalSince1970))

var nonce = randomStringWithLength(11)

var baseUrl: String

if signUrl == nil {

baseUrl = (request.valueForKey("URL") as! NSURL).absoluteString!

print("REQUEST URL: " + baseUrl)

}

else {

baseUrl = signUrl

print("SIGN URL: " + signUrl)

}

print("TIMESTAMP: " + timeStamp)

print("NONCE: " + nonce)

To calculate the signature we follow the guidance from the OAuth 1.0a specs, and we will find that the Fitbit API is following them indeed to the letter. In fact, we built this initially using the Google interactive OAuth 1.0a page, and it just worked with Fitbit. The order of the params has to be preserved exactly as shown. We also log in to the console the elements that contribute to the signature, to be able to verify the output with the API support page. It is important to observe that we escape both the baseUrl and the normalizedParameters strings once more before joining them into the signatureBaseString. We also notice that the urlParameters are mixed with the OAuth parameters; in fact, the resulting URL string needs to be sorted alphabetically by keys. The sorting is case-sensitive, so Z comes before a.

The signing key for hmac() is the concatenated values (each first encoded per Parameter Encoding) of the Consumer Secret and Token Secret, separated by an ‘&’ character (ASCII code 38) even if empty.

Listing 5-27.

var signatureParams: [String:String] = [:]

for (key, value) in urlParameters {

signatureParams.updateValue(value, forKey: key)

}

signatureParams.updateValue(oauthConsumerKey, forKey: "oauth\_consumer\_key")

signatureParams.updateValue(nonce, forKey: "oauth\_nonce")

signatureParams.updateValue(signatureMethod, forKey: "oauth\_signature\_method")

signatureParams.updateValue(timeStamp, forKey: "oauth\_timestamp")

if oauthToken != nil {

signatureParams.updateValue(oauthToken, forKey: "oauth\_token")

request.setValue(oauthToken, forHTTPHeaderField: "oauth\_token")

}

signatureParams.updateValue(oauthVersion, forKey: "oauth\_version")

var normalizedParameters: String = asURLString(inputData: signatureParams)

var signatureBaseString: String = "&".join([

request.HTTPMethod,

baseUrl.escapeUrl(),

normalizedParameters.escapeUrl()

])

// the key is the concatenated values (each first encoded per Parameter Encoding)

// of the Consumer Secret and Token Secret, separated by an ‘&’ character (ASCII code 38) even if empty

var signKey = oauthConsumerSecret.escapeUrl() + "&" + oauthTokenSecret.escapeUrl()

var signature = signatureBaseString.hmac(HMACAlgorithm.SHA1, key: signKey)

print("SIGNATURE STRING: " + signatureBaseString)

print("SIGNATURE KEY: " + signKey)

print("SIGNATURE: " + signature)

Now that we have the signature, we need to populate the request header with the correct string. Here, too, the alphabetic order of the parameters is important.

Listing 5-28.

// This exact order has to be preserved  
 let header: OAuth1aHeader = OAuth1aHeader(name: "OAuth")  
 header.add("oauth\_consumer\_key", value: oauthConsumerKey)  
 header.add("oauth\_nonce", value: nonce)  
 header.add("oauth\_signature", value: signature)  
 header.add("oauth\_signature\_method", value: signatureMethod)  
 header.add("oauth\_timestamp", value: timeStamp)  
 header.add("oauth\_token", value: oauthToken)  
 header.add("oauth\_version", value: oauthVersion)  
 let hParams = header.asString()  
   
 print("HEADER: Authorization: " + hParams)  
 request.setValue(hParams, forHTTPHeaderField: "Authorization")

### Creating the OAuth header

The code that puts together correctly the header entry and escapes the values is tucked away in an inner class, at the end of the OAuth1a.swift file:

Listing 5-29.

class OAuth1aHeader {

var hName: String!

var params: Array<String>!

required init (name: String) {

params = Array<String>()

hName = name

}

func add (key: String, value: String) {

params.append(key + "=\"" + value.escapeUrl() + "\"")

}

func asString () -> String {

var hParams: String = ", ".join(params)

return hName + " " + hParams

}

}

We did not create a common library for the APIClient and this OAuth1a library, so we need to duplicate some code that encodes the URL for signing. Sorting of parameters was not needed in the actual URL but is needed for the correct OAuth signature.

Listing 5-30.

func asURLString (inputData: [String:String]!=[:]) -> String {

var params: [String] = []

for (key, value) in inputData {

params.append( "=".join([ key.escapeUrl(), value.escapeUrl()] ))

}

params = params.sort { $0 < $1 }

return "&".join(params)

}

This was all the work needed for the OAuth signature. It wasn’t that bad, right? As a bonus, we added the signTempAccessToken() function that can be used for the signing of the first step in registering the access of your app to a user account. Notice that the requestUrl is stripped of the protocol and the host part, so that instead of the full URL https://api.fitbit.com/oauth/request\_token only the oauth/request\_token path is used for signing.

Listing 5-31.

func signTempAccessToken (request: NSMutableURLRequest) {

// This request does not use the URL for signing, but rather the path oauth/request\_token

var requestUrl = request.valueForKey("URL") as? NSURL

var urlPath: String = requestUrl!.path!

urlPath = String( dropFirst(urlPath) )

signRequest(request, signUrl: urlPath)

}

### The code for OAuth1a.swift

Here is the complete code in the OAuth1a.swift library.

Listing 5-32.

import Foundation

class OAuth1a {

var signatureMethod: String = "HMAC-SHA1"

var oauthVersion: String = "1.0"

var oauthConsumerKey: String!

var oauthConsumerSecret: String!

var oauthToken: String!

var oauthTokenSecret: String!

required init (oauthParams: NSDictionary) {

oauthConsumerKey = oauthParams.objectForKey("oauth\_consumer\_key") as! String

oauthConsumerSecret = oauthParams.objectForKey("oauth\_consumer\_secret") as! String

oauthToken = oauthParams.objectForKey("oauth\_token") as! String

oauthTokenSecret = oauthParams.objectForKey("oauth\_token\_secret") as! String

}

func randomStringWithLength (len : Int) -> String {

let letters : NSString = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

var randomString : NSMutableString = NSMutableString(capacity: len)

for (var i=0; i < len; i++){

var length = UInt32(letters.length)

var rand = arc4random\_uniform(length)

randomString.appendFormat("%C", letters.characterAtIndex(Int(rand)))

}

return randomString as String

}

func signRequest (request: NSMutableURLRequest, urlParameters: [String:String]!=[:], signUrl: String!=nil) {

let timeStamp = String(format:"%d", Int(NSDate().timeIntervalSince1970))

var nonce = randomStringWithLength(11)

var baseUrl: String

if signUrl == nil {

baseUrl = (request.valueForKey("URL") as! NSURL).absoluteString!

print("REQUEST URL: " + baseUrl)

}

else {

baseUrl = signUrl

print("SIGN URL: " + signUrl)

}

print("TIMESTAMP: " + timeStamp)

print("NONCE: " + nonce)

// The signing params need to be sorted alphabetically

var signatureParams: [String:String] = [:]

for (key, value) in urlParameters {

signatureParams.updateValue(value, forKey: key)

}

signatureParams.updateValue(oauthConsumerKey, forKey: "oauth\_consumer\_key")

signatureParams.updateValue(nonce, forKey: "oauth\_nonce")

signatureParams.updateValue(signatureMethod, forKey: "oauth\_signature\_method")

signatureParams.updateValue(timeStamp, forKey: "oauth\_timestamp")

if oauthToken != nil {

signatureParams.updateValue(oauthToken, forKey: "oauth\_token")

request.setValue(oauthToken, forHTTPHeaderField: "oauth\_token")

}

signatureParams.updateValue(oauthVersion, forKey: "oauth\_version")

var normalizedParameters: String = asURLString(inputData: signatureParams)

var signatureBaseString: String = "&".join([

request.HTTPMethod,

baseUrl.escapeUrl(),

normalizedParameters.escapeUrl()

])

// the key is the concatenated values (each first encoded per Parameter Encoding)

// of the Consumer Secret and Token Secret, separated by an ‘&’ character (ASCII code 38) even if empty

var signKey = oauthConsumerSecret.escapeUrl() + "&" + oauthTokenSecret.escapeUrl()

var signature = signatureBaseString.hmac(HMACAlgorithm.SHA1, key: signKey)

print("SIGNATURE STRING: " + signatureBaseString)

print("SIGNATURE KEY: " + signKey)

print("SIGNATURE: " + signature)

// This exact order has to be preserved

let header: OAuth1aHeader = OAuth1aHeader(name: "OAuth")

header.add("oauth\_consumer\_key", value: oauthConsumerKey)

header.add("oauth\_nonce", value: nonce)

header.add("oauth\_signature", value: signature)

header.add("oauth\_signature\_method", value: signatureMethod)

header.add("oauth\_timestamp", value: timeStamp)

header.add("oauth\_token", value: oauthToken)

header.add("oauth\_version", value: oauthVersion)

let hParams = header.asString()

print("HEADER: Authorization: " + hParams)

request.setValue(hParams, forHTTPHeaderField: "Authorization")

}

func asURLString (inputData: [String:String]!=[:]) -> String {

var params: [String] = []

for (key, value) in inputData {

params.append( "=".join([ key.escapeUrl(), value.escapeUrl()] ))

}

params = params.sorted{ $0 < $1 }

return "&".join(params)

}

func signTempAccessToken (request: NSMutableURLRequest) {

// This request does not use the URL for signing, but rather the path oauth/request\_token

var requestUrl = request.valueForKey("URL") as? NSURL

var urlPath: String = requestUrl!.path!

urlPath = String( dropFirst(urlPath) )

signRequest(request, signUrl: urlPath)

}

class OAuth1aHeader {

var hName: String!

var params: Array<String>!

required init (name: String) {

params = Array<String>()

hName = name

}

func add (key: String, value: String) {

params.append(key + "=\"" + value.escapeUrl() + "\"")

}

func asString () -> String {

var hParams: String = ", ".join(params)

return hName + " " + hParams

}

}

## Testing what we have so far

With the code we have so far, we can make requests to the local host, provided that we set up the local playground with Apache and we have the two test documents in place. We clicked the Good Request once the Bad Request once, and in Listing 5-33 we can see the Xcode console output of the println() statements.

Listing 5-33.

REQUEST URL: http://127.0.0.1/data.json

TIMESTAMP: 1429481277

NONCE: OPrkKRdgn8I

SIGNATURE STRING: GET&http%3A%2F%2F127.0.0.1%2Fdata.json&oauth\_consumer\_key%3D6cf4162a72ac4a4382c098caec132782%26oauth\_nonce%3DOPrkKRdgn8I%26oauth\_signature\_method%3DHMAC-SHA1%26oauth\_timestamp%3D1429481277%26oauth\_token%3D5a3ca2edf91d7175cad30bc3533e3c8a%26oauth\_version%3D1.0

SIGNATURE KEY: c652d5fb28f344679f3b6b12121465af&da5bc974d697470a93ec59e9cfaee06d

SIGNATURE: 2Efcl/SN9s+xR9qRTObIsNQwkpI=

HEADER: Authorization: OAuth oauth\_consumer\_key="6cf4162a72ac4a4382c098caec132782", oauth\_nonce="OPrkKRdgn8I", oauth\_signature="2Efcl%2FSN9s%2BxR9qRTObIsNQwkpI%3D", oauth\_signature\_method="HMAC-SHA1", oauth\_timestamp="1429481277", oauth\_token="5a3ca2edf91d7175cad30bc3533e3c8a", oauth\_version="1.0"

RESPONSE RAW: {"Response":{"key":"value"}}

RESPONSE SHA1: 5ae11e3b34fdcd7fba984695e9001511c9e0aa8d

REQUEST URL: http://127.0.0.1/badData.json

TIMESTAMP: 1429481278

NONCE: 0OHgLcjefM6

SIGNATURE STRING:

GET&http%3A%2F%2F127.0.0.1%2FbadData.json&oauth\_consumer\_key%3D6cf4162a72ac4a4382c098caec132782%26oauth\_nonce%3D0OHgLcjefM6%26oauth\_signature\_method%3DHMAC-SHA1%26oauth\_timestamp%3D1429481278%26oauth\_token%3D5a3ca2edf91d7175cad30bc3533e3c8a%26oauth\_version%3D1.0

SIGNATURE KEY: c652d5fb28f344679f3b6b12121465af&da5bc974d697470a93ec59e9cfaee06d

SIGNATURE: DMB81a3oUO16lJa7YvUJpYguunQ=

HEADER: Authorization: OAuth oauth\_consumer\_key="6cf4162a72ac4a4382c098caec132782", oauth\_nonce="0OHgLcjefM6", oauth\_signature="DMB81a3oUO16lJa7YvUJpYguunQ%3D", oauth\_signature\_method="HMAC-SHA1", oauth\_timestamp="1429481278", oauth\_token="5a3ca2edf91d7175cad30bc3533e3c8a", oauth\_version="1.0"

RESPONSE RAW: {"Response”:{{“key":"value"}}

RESPONSE SHA1: bd28faef1bc309899ed8540105c86ad23c1f27e7

Our simulator screenshot in Figure 5-4 shows the results of our activity. Now we are ready to test against the live API, and see if our OAuth work paid off.

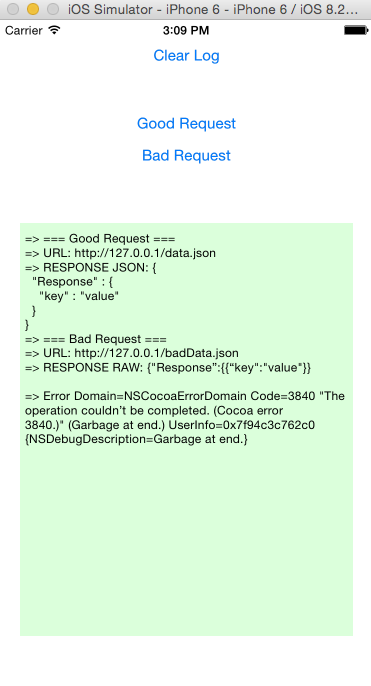


Figure 5-4. The simulator screenshot

# Making requests to the Fitbit API

By now we know that there is no magic to the API request process; it is just important to assemble all the parts of the request in the correct order.

* Get the correct tokens and create the OAuth signature with all elements in the correct order.
* Keep track of the rate limiting parameters that could cripple your app if there is too much traffic.
* Build a parser that handles the JSON output.

On the Fitbit developer page, we can use the debug tool to place an API request, now that we know our access token and access token secret. Keep in mind that the request is signed not only with the access token and the access token secret but also with the oauth\_consumer\_key, which will be used by the API to identify the user account being accessed. The oauth\_consumer\_key is present in the Header, while the oauth\_consumer\_secret is used together with oauth\_token\_secret to sign the Base String.

Listing 5-34shows the dissected request for the user profile, when we make the request using curl. Shown are the defaults for a profile that has not been updated since the creation of the account. The format of the response is well documented on the Fitbit developer site.

Listing 5-34.

Access Token: 5a3ca2edf91d7175cad30bc3533e3c8a

Access Token Secret: da5bc974d697470a93ec59e9cfaee06d

Request URL: https://api.fitbit.com/1/user/-/profile.json

Nonce: random

Timestamp: 1429396457

Type: GET

API request values:

Base string: GET&https%3A%2F%2Fapi.fitbit.com%2F1%2Fuser%2F-%2Fprofile.json&oauth\_consumer\_key%3D6cf4162a72ac4a4382c098caec132782%26oauth\_nonce%3Drandom%26oauth\_signature\_method%3DHMAC-SHA1%26oauth\_timestamp%3D1429396457%26oauth\_token%3D5a3ca2edf91d7175cad30bc3533e3c8a%26oauth\_version%3D1.0

Signed with: c652d5fb28f344679f3b6b12121465af&da5bc974d697470a93ec59e9cfaee06d

Signature: c2wi9Xk+nOGpjRoyxtotIM5AyA4=

Listing 5-35 shows the request made with curl.

Listing 5-35.

$ curl -X GET -i -H 'Authorization: OAuth oauth\_consumer\_key="6cf4162a72ac4a4382c098caec132782", oauth\_nonce="random", oauth\_signature="c2wi9Xk%2BnOGpjRoyxtotIM5AyA4%3D", oauth\_signature\_method="HMAC-SHA1", oauth\_timestamp="1429396457", oauth\_token="5a3ca2edf91d7175cad30bc3533e3c8a", oauth\_version="1.0"' https://api.fitbit.com/1/user/-/profile.json

Listing 5-36 shows the output of the curl request.

Listing 5-36.

HTTP/1.1 200 OK

Server: nginx

X-UA-Compatible: IE=edge,chrome=1

Expires: Thu, 01 Jan 1970 00:00:00 GMT

Cache-control: no-cache, must-revalidate

Pragma: no-cache

Fitbit-Rate-Limit-Limit: 150

Fitbit-Rate-Limit-Remaining: 149

Fitbit-Rate-Limit-Reset: 1478

Set-Cookie: JSESSIONID=5D7EA76F7CB0C45BF1A7A020C9BAC55B.fitbit1; Path=/; HttpOnly

Content-Type: application/json;charset=UTF-8

Content-Language: en

Content-Length: 657

Vary: Accept-Encoding

Date: Sat, 18 Apr 2015 22:35:21 GMT

{"user":{"avatar":"http://www.fitbit.com/images/profile/defaultProfile\_100\_male.gif","avatar150":"http://www.fitbit.com/images/profile/defaultProfile\_150\_male.gif","country":"US","dateOfBirth":"","displayName":"","distanceUnit":"en\_US","encodedId":"3BRQLQ","foodsLocale":"en\_US","gender":"NA","glucoseUnit":"en\_US","height":0,"heightUnit":"en\_US","locale":"en\_US","memberSince":"2015-04-02","offsetFromUTCMillis":-25200000,"startDayOfWeek":"SUNDAY","strideLengthRunning":86.60000000000001,"strideLengthWalking":67.10000000000001,"timezone":"America/Los\_Angeles","topBadges":[],"waterUnit":"en\_US","waterUnitName":"fl oz","weight":62.5,"weightUnit":"en\_US"}}

To make live requests, we need to uncomment the api.goLive() in the ViewController library, and our requests will go directly to the Fitbit API. Of course, you need to make sure that the oauthParams in the APIClient are set to the values you generated while going through the registration steps of your application, as shown in section “The Fitbit OAuth Implementation.”

## Retrieving the user profile

Not to forget, we have two test buttons in the view controller that test against the local documents. The easiest way to test if the API is working is to repurpose the code and make a user profile request. In the ViewController.swift file we have the following:

@IBAction func clickButton() {

logger.logEvent("=== Good Request ===")

// api.getData(APIService.GOOD\_JSON) // TEST CALL

api.getData(APIService.USER, id: "-", urlSuffix: NSArray(array: ["profile"]))

labelButton.setTitle("Good Request Sent", forState: UIControlState.Normal)

}

We click the button and here is our first API call made through the application works. In the Xcode console we see the full detail of the request/response (see Listing 5-7).

Listing 5-37.

REQUEST URL: https://api.fitbit.com/1/user/-/profile.json

TIMESTAMP: 1429482687

NONCE: eTYGnygDYr4

SIGNATURE STRING: GET&https%3A%2F%2Fapi.fitbit.com%2F1%2Fuser%2F-%2Fprofile.json&oauth\_consumer\_key%3D6cf4162a72ac4a4382c098caec132782%26oauth\_nonce%3DeTYGnygDYr4%26oauth\_signature\_method%3DHMAC-SHA1%26oauth\_timestamp%3D1429482687%26oauth\_token%3D5a3ca2edf91d7175cad30bc3533e3c8a%26oauth\_version%3D1.0

SIGNATURE KEY: c652d5fb28f344679f3b6b12121465af&da5bc974d697470a93ec59e9cfaee06d

SIGNATURE: a+CWGxlsJiiJGc7ezIZVNtw3ASA=

HEADER: Authorization: OAuth oauth\_consumer\_key="6cf4162a72ac4a4382c098caec132782", oauth\_nonce="eTYGnygDYr4", oauth\_signature="a%2BCWGxlsJiiJGc7ezIZVNtw3ASA%3D", oauth\_signature\_method="HMAC-SHA1", oauth\_timestamp="1429482687", oauth\_token="5a3ca2edf91d7175cad30bc3533e3c8a", oauth\_version="1.0"

RESPONSE HEADER rateLimit: 150

RESPONSE HEADER rateLimitRemaining: 149

RESPONSE HEADER rateLimitReset: 1713, checked at: 1429482687

RESPONSE RAW: {"user":{"avatar":"http://www.fitbit.com/images/profile/defaultProfile\_100\_male.gif","avatar150":"http://www.fitbit.com/images/profile/defaultProfile\_150\_male.gif","country":"US","dateOfBirth":"","displayName":"","distanceUnit":"en\_US","encodedId":"3BRQLQ","foodsLocale":"en\_US","gender":"NA","glucoseUnit":"en\_US","height":0,"heightUnit":"en\_US","locale":"en\_US","memberSince":"2015-04-02","offsetFromUTCMillis":-25200000,"startDayOfWeek":"SUNDAY","strideLengthRunning":86.60000000000001,"strideLengthWalking":67.10000000000001,"timezone":"America/Los\_Angeles","topBadges":[],"waterUnit":"en\_US","waterUnitName":"fl oz","weight":62.5,"weightUnit":"en\_US"}}

RESPONSE SHA1: caa14550567548b173a0a904f0b3c7bae6d7452a

Figure 5-5 shows the screenshot of the user profile as it was read.

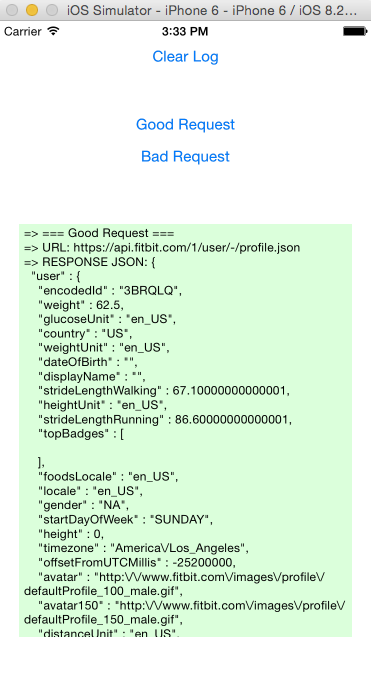


Figure 5-5. Screenshot of the user profile

## Retrieving and setting data in the API

Like retrieving the user profile, this is a simple GET with a given API target. Following is a nonexhaustive list of targets you can currently query that retrieve and set user data:

* Get/Set body measurements
* Get/Set body weight and body fat
* Get/Set blood pressure and heart rate
* Get/Set glucose

With the exception of body weight and body fat, all the foregoing targets are unfortunately on the list of deprecated API features (as of October 2014). We expect that ultimately all services will be migrated to OAuth 2.0. Deprecated means that the features are still available but are not actively developed and will be later replaced by other features. The deprecated services will still be available for a while, after the new services are in place.

Fitbit announced three new products at the time of writing of this book: Charge, Charge HR, and Surge. The Fitbit API will provide access to the all-day heart rate and GPS data from these devices; however, these data types will be accessible exclusively via OAuth 2.0.

As the transition to OAuth 2.0 will be happening in the coming months, there is little point in detailing here how to use the entire set of deprecated API services. To make things easy we will show how to get and set a few data points: all other services are similar in nature to the request response, and well documented on the Fitbit developer page.

### Getting the Blood Pressure

We will repurpose our “Good Request” Button in the ViewController.swift file to make this request:

@IBAction func clickButton() {

logger.logEvent("=== Good Request ===")

// api.getData(APIService.GOOD\_JSON) // TEST CALL

// api.getData(APIService.USER, id: "-", urlSuffix: NSArray(array: ["profile"]))

api.getBloodPressure()

labelButton.setTitle("Good Request Sent", forState: UIControlState.Normal)

}

The function we wrote in the APIClient is rather simple, and it makes use of the abstraction we created with getData(). This method is located in the APIClient.swift file.

func getBloodPressure (date: NSDate?=NSDate()) {

let formatter = NSDateFormatter()

formatter.dateFormat = "yyyy-MM-dd"

let currentDate = formatter.stringFromDate(date!)

print("CURRENT DATE: \(currentDate)");

getData(APIService.USER, id: "-", urlSuffix: NSArray(array: ["bp/date", currentDate ]))

}

The function makes the following API call:

GET /1/user/-/bp/date/2015-04-23.json

Naturally, since we did not send any blood pressure information to the API yet, the default response is blank, as shown below. Your response handler needs to be able to deal with an empty response.

{"bp":[]}

Keep in mind that this response right now goes nowhere. When you implement your application, you will have decide how to handle it in the processGETData() callback function in the API, based on the data you set in the urlSuffix param, which we now know can be composed of the service name and its parameters.

When there are records for the blood pressure in the system, our response will be more like the following example shown in Listing 5-38:

Listing 5-38.

{

    "average": {

            "condition":"Prehypertension",

            "diastolic":85,

            "systolic":115

    },

    "bp": [

        {

            "diastolic":80,

            "logId":483697,

            "systolic":120

        },

        {

            "diastolic":90,

            "logId":483699,

            "systolic":110,

            "time":"08:00"

        }

    ]

}

### Setting the Blood Pressure

Continuing to abuse the “Good Request” Button, we change things once again, this time calling the new function. Keep in mind that these are both async methods, and the way we implement them here is not necessarily the best way, as there is no guarantee which one reaches the API first, and they both rely on the processGETData() in the APIClient to handle the response. This function is in ViewController.swift.

@IBAction func clickButton() {  
 logger.logEvent("=== Good Request ===")  
 // api.getData(APIService.GOOD\_JSON) // TEST CALL  
 api.getData(APIService.USER, id: "-", urlSuffix: NSArray(array: ["profile"]))  
 labelButton.setTitle("Good Request Sent", forState: UIControlState.Normal) }

We keep the getBloodPressure() function we wrote before, since we will need to read back the data we are setting so we create the setBloodPressure() function(located in APIClient.swift):

func setBloodPressure (date: NSDate?=NSDate()) {

let formatter = NSDateFormatter()

formatter.dateFormat = "yyyy-MM-dd"

let currentDate = formatter.stringFromDate(date!)

let request: [String:String] = ["diastolic":"80","systolic":"120","date": currentDate]

postData(APIService.USER, id: "-", urlSuffix: NSArray(array: ["bp" ]), params: request)

}

This function makes the following API call:

POST /1/user/-/bp.json?date=2015-04-23&diastolic=80&systolic=120

The API responds with the data that was just inserted, with a populated logId.

{

"bpLog": {

"diastolic":80,

"logId":1298241959,

"systolic":120

}

}

If we now make a call to get the blood pressure data, we get the full detail, including the friendly health warning. It is important to note that there is no timestamp, because we did not provide one, the field being optional.

{

"average": {

"condition":"Prehypertension",

"diastolic":80,

"systolic":120

},

"bp": [

{

"diastolic":80,

"logId":1298241959,

"systolic":120

}

]

}

### Logging the Body Weight

Just as we did for blood pressure, the service that lets you set and get body weight data has a similar implementation. To set the body weight, we call the service with the parameters listed in the API docs, very much like we did with the other service. Save this function in the APIClient.swift.

func setBodyWeight (date: NSDate?=NSDate()) {

let formatter = NSDateFormatter()

formatter.dateFormat = "yyyy-MM-dd"

let currentDate = formatter.stringFromDate(date!)

let

postData(APIService.USER, id: "-", urlSuffix: NSArray(array: ["body/log/weight" ]), params: request)

}

This function makes the following API call:

POST /1/user/-/bp.json?date=2015-04-24&weight=73

The GET is no different from the blood pressure call, other than the URL path(APIClient.swift):

func getBodyWeight (date: NSDate?=NSDate()) {

let formatter = NSDateFormatter()

formatter.dateFormat = "yyyy-MM-dd"

let currentDate = formatter.stringFromDate(date!)

getData(APIService.USER, id: "-", urlSuffix: NSArray(array: ["body/log/weight/date", currentDate ]))

}

This function makes the following API call:

GET /1/user/-/body/log/weight/date/2015-04-24.json

The response from API is JSON data in the following format:

{

"weight": [

{

"bmi":0,

"date":"2015-04-24",

"logId":1429919999000,

"time":"23:59:59",

"weight":73

}

]

}

## OAuth versions: Working in both worlds

The current version of the API used by Fitbit is OAuth 1.0a. This is a stable, very secure, and reliable protocol but intrinsically complex. Just as is the case with PGP encryption, complex and secure crypto and protocols are being replaced by others that might not be so good, but they are convenient to implement.

OAuth 1.0a is signing every request, and that can be a resource drain on either side of the implementation (client/server). At the same time, there is no clear separation of roles between the authorization server and the resource server since a lot of data needed for signing is common to these roles. Granted, OAuth 1.0a has its place for the security it offers; alone for the sheer advantage of being able to sign and trust every request, it might never completely go away, but ease of implementation is not its forte. Consider, however, that with OAuth 1.0a when correctly implemented, the chance of success for a Man-in-the-Middle type of attack is extremely low, while with token-based protocols like OAuth 2.0, it’s a simple matter of breaking SSL, which in the last years is developing holes much like Swiss cheese.

A lot of the recent API implementations favor OAuth 2.0.

OAuth 2.0 also accommodates much easier native applications with specifically suited workflows. It also provides a very clear separation of roles between the authentication server and the server handling the request.

Viewed on an elementary level, OAuth 2.0 defines the following workflow:

* Authenticate with the authorization server, and get an authorization code
* Request from the authorization server a set of access and refresh tokens
* Using the access token, request restricted resources from the resource server
* Periodically use the refresh token to get from the authorization server a new access token

The access token is set to have an expiration date, and the lifetime of one token varies with the implementation.

Given that OAuth 2.0 also gets a bad rap from security experts for the many loose ends in the specification and the potential for abuse and misrepresentation of the caller, it is conceivable that further versions of the standard will come up in the next years. For that matter alone, it would serve you well to follow the development of the standard and how service providers like Fitbit follow up with it.

In the case of Fitbit, the company announced support for OAuth 2.0, but there was no clear detail of implementation available at the time of writing this chapter. If you need to deliver your application right now, you should write services following the current OAuth 1.0a standard, but also be ready to provide upgrades to your app that will use the OAuth 2.0 services when they become public and well documented.

Consider that we went through a rather complex process to implement (in part) the OAuth 1.0a for the Fitbit API in its current version; looking back at the code, it wasn’t all that bad. Writing code for OAuth 2.0 is likely to be easier, at least because you won’t need to sign every request, and you can reuse much of this code: in fact you will have to, because for a good while you will have to live in both worlds, as Fitbit transitions the API from OAuth 1.0a to OAuth 2.0.

As mentioned at the beginning of this chapter, the transition from old to new is a gradual process for most APIs. Your application will be alive and well for quite a while, and you will have the time to study, implement, test, and deliver an upgraded version of your app that supports OAuth 2.0. What is most likely to happen is that Fitbit will offer specific services only on OAuth 2.0; so, for that reason alone, you will have to upgrade your app to support it as soon as it becomes available.

# Summary

We learned in this chapter how to build an API to communicate with the Fitbit API and implement a few calls that get and set health data. Since this API, as most APIs out there, is under constant development, you might need to implement new or different calls to the API to do what we did here, as well as add support for OAuth 2.0 for the new API version, but the basic principles remain the same.