Chapter 12

Building an App That Interacts   
with a Raspberry Pi

# Gheorghe Chesler

In this chapter we will write an app that communicates with a Raspberry Pi device on the local wifi network and allows us to flip the lights on and off on a custom module with LEDs. This might not seem like a significant accomplishment, but keep in mind that just as you can flip on and off some LED lights, you can control any other connected devices in a similar fashion.

# About Your Raspberry Pi

Raspberry Pi is a miniaturized ARM computer that can run different operating systems. By default, this would run a variant of Debian Linux customized for this device, called Raspbian. On the Raspberry Pi 2 you can also install Ubuntu, RiscOS, and recently even a slimmed-down, custom version of Windows 10. Of course, we need to keep in mind that the CPU is not very powerful, and the Raspberry Pi device does not have a lot of memory: the B version has 512MB RAM, and the latest version (Pi 2) has 1GB of RAM, which happens to be the maximum memory supported by the ARM7 CPU. This is definitely not enough memory to do any fancy computing: as of the writing of this book there is no easy way to install a popular browser. The chromium browser is supported, but Google Chrome is not yet available: even if it were, it is too big for the available system resources to do anything you are accustomed to on a regular computer.

What the Raspberry Pi loses in terms of computing power it makes up with the very low power usage and the ability to connect smart devices and accessories like LED lights, relays, motors, camera, pressure, acceleration, and humidity sensors, to name just a few.

Another factor is the price: the regular Pi model B is available for about $30, and you can get most extension modules at really accessible prices from a lot of online and local retailers.

The Pi 2 Model B is based around a Broadcom BCM2836 SoC, which includes a Quad Core ARM7 900 MHz processor, 1GB of RAM, and 4 USB ports where you can plug in external devices. The Pi also has an Ethernet connector, a HDMI connector for the display, and an audio port. The older Pi model B+ uses a less powerful, single core Broadcom BCM2835 ARM11 700MHz processor and has only 512MB of RAM, while the very first model came only with 256MB of RAM.

To power the Raspberry Pi you will need to connect it via a micro-USB to any wall charger or to a computer that provides enough power for the Pi and the connected devices. In most cases 1A would be sufficient, which is what the regular USB ports provide. To the power needs of the main board you have to add the current consumed by your plug-in boards. The Pi does not include a built-in hard disk or solid-state drive, instead relying on a microSD card for booting and long-term storage.

The Raspberry Pi does not come with a real-time clock, so an OS must use a network time server, or ask the user for time information at boot time to get access to time and date info for file time and date stamping. However, a real-time clock (such as the DS1307) with battery backup can be easily added via the I2C interface.

The Raspberry Pi also provides two onboard ribbon slots to connect a camera and a display. Most plastic cases that you would buy to host your Raspberry Pi provide holes for guiding these cables outside the enclosure.

For the scope of this chapter, we picked a very simple module that has a number of LEDs on a miniature board, along with the chip that controls them. The module is named "PiGlow" and is marketed by the company Pimoroni:

https://shop.pimoroni.com/products/piglow

In Figure 12-1 we can see our Raspberry Pi B+ with a PiGlow board attached.



Figure 12-1. Raspberry Pi B+ with a PiGlow board attached

If you have an older Raspberry Pi you will still be able to do everything we do in this chapter: the only things that the new model gets you are a faster processor and more memory. Some of the commands we will run are specific to the model of Raspberry Pi you own: inline comments will specify these differences.

# Control interfaces on your Raspberry Pi

Each embedded platform has one or more ways to access connected or native devices on the local hardware.

A very common one is GPIO (General Purpose Input Output). This allows you to flip a light or a relay on or off, and for that purpose it is really great, but there is a limit to how many devices can be addressed (more or less limited by the number of pins available on the custom connector).

The I2C (Inter IC) interface gives you more flexibility in what you can do with individual devices, and of course it allows you to attach more devices.

The I2C bus was designed by Philips in the early 1980s to allow easy communication between components that reside on the same circuit board. For a complete reference on the I2C interface, see the following URL: www.i2c-bus.org/i2c-bus/.

I2C uses 7 bit addressing. The maximum number that can be expressed with 7 bits is 128, which means you can have more than 120 devices you can access via the same bus. I2C is also much simpler to implement in hardware—it requires only two connection lines, one for the clock and the other for the data.

# Setting up your Raspberry Pi

The easiest way to get started with Raspberry Pi is to install the Raspbian operating system. The Raspberry Pi official web page has it available for download here: www.raspberrypi.org/downloads/.

To make things easy, use the NOOBS (New Out-Of-Box Software) zip file.

On the same page there are also installation instructions; after unpacking the downloaded zip file, you format a reasonably large micro-SD card (4GB or more) with the FAT file system, and copy the files on the card. Then, insert the card in your Raspberry Pi device and start it. When the device boots, it will prompt you with the choice of OS and language, and you can get it installed.

For the installation phase you can use an HDMI monitor or a VGA monitor via an HDMI to VGA adapter; this will not be needed later on, because you can SSH (secure shell) into the device from another computer.

To SSH into the Raspberry Pi device from another computer you can use the following credentials:

user: pi

password: raspberry

If you did not install the GUI (graphical user interface) by default, you can start the GUI at any time using the following command:

startx

To configure the device and ports you can use the Raspberry Pi configuration utility:

raspi-config

After you installed the OS, the first step is to bring the system up to date. For that, you can enter the following in your SSH window:

sudo apt-get update

sudo apt-get upgrade

To force a command to be executed with root privileges we use the sudo command in front of a regular command. This is necessary because the standard "pi" user does not have enough rights to modify system files or install applications.

The apt-get utility is the standard Debian package manager, which you might be familiar with if you are using Ubuntu. The first command fetches the list of the latest available packages, while the second one installs updates available for the currently installed system modules/packages.

## Choosing the scripting language

On the Raspberry Pi, Python has become the de facto standard scripting language, because there are so many packages made available to control every device imaginable. This being said, you have to keep in mind that most of them will be Python 2, and some will be Python 3. The Python language has a split personality: the Python 2 is the older, more established version, with wider support. Python 3 is the new, “better” version that offers modern language constructs for object oriented design, but the support for it on platforms such as Raspberry Pi is limited, and the scripts and packages written for Python 3 will not work for Python 2.

At the same time, there is a considerably large support of Perl on embedded platforms; however, installing some of the Perl packages could be a daunting task that most novices would have a hard time completing.

## Configuring I2C

You can find most of the instructions in this section at the following URL, where you can also find more tutorials on how to install other accessories and plug-ins for your Raspberry Pi:

https://learn.adafruit.com/adafruits-raspberry-pi-lesson-4-gpio-setup/configuring-i2c

I2C is a very commonly used standard designed to allow one chip to talk to another.

Since the Raspberry Pi can talk to I2C, we can connect it to a variety of I2C-capable chips and modules.

Following are some of the Adafruit projects that make use of I2C devices and modules:

* http://learn.adafruit.com/mcp230xx-gpio-expander-on-the-raspberry-pi
* http://learn.adafruit.com/adafruit-16x2-character-lcd-plus-keypad-for-raspberry-pi
* http://learn.adafruit.com/adding-a-real-time-clock-to-raspberry-pi
* http://learn.adafruit.com/matrix-7-segment-led-backpack-with-the-raspberry-pi
* http://learn.adafruit.com/mcp4725-12-bit-dac-with-raspberry-pi
* http://learn.adafruit.com/adafruit-16-channel-servo-driver-with-raspberry-pi
* http://learn.adafruit.com/using-the-bmp085-with-raspberry-pi

The I2C bus allows multiple devices to be connected to your Raspberry Pi, each with a unique address. The device address can often be set by changing jumper settings on the module. It is very useful to be able to see which devices are connected to your Pi as a way of making sure everything is working. To do this, it is worth running the following command in the Terminal to install the i2c-tools utility:

sudo apt-get install i2c-tools

To keep things simple, we will use Python for our application we write for the Raspberry Pi. To use Python with I2C tools, we need to install the python-smbus package:

sudo apt-get install python-smbus

If you plan to use Perl to build tools that control the I2C devices, you will need to install some extra packages that provide an interface to the I2C tools to Perl packages, as well as some basic framework that allows writing compact applications:

sudo apt-get install libi2c-dev build-essential libmoose-perl

sudo cpan Device::SMBus

### Installing Kernel Support for I2C

Run the raspi-config as root and follow the prompts to install i2c support for the ARM core and linux kernel:

sudo raspi-config

Select Advanced Options/I2C and enable the interface, as well as allow the I2C kernel module to be loaded by default. After you made the changes you have to reboot the device, and the modules should be loaded and available after reboot.

Verify that the i2c modules are available, by looking at the content of the following file:

cat /etc/modules

The file will have contents similar to the following example:

# /etc/modules: kernel modules to load at boot time.

#

# This file contains the names of kernel modules that should be loaded

# at boot time, one per line. Lines beginning with "#" are ignored.

i2c-bcm2708

i2c-dev

If after reboot, these modules do not exist in the /etc/modules file: you can edit the file with vi and append the two previously mentioned lines. You have to edit this file with root privileges:

sudo vi /etc/modules

Another easy way to do this is with the following one-liners:

sudo echo "i2c-bcm2708" >> /etc/modules

sudo echo "i2c-dev" >> /etc/modules

Depending on when you installed your OS and with what version, as well as on any other experiments you might have run with your Raspberry Pi, some of these modules could have landed in the modprobe blacklist. This is a file that disables certain modules, preventing them to be loaded at start. To look at the file contents, see:

sudo cat /etc/modprobe.d/raspi-blacklist.conf

The same as we did with the /etc/modules, you can edit the file with vi and comment out or delete the lines if they exist in that file. Here too, you have to use root privileges to edit the file:

sudo vi /etc/modprobe.d/raspi-blacklist.conf

There is a fancier way to do just about the same thing with the following one-liner:

MPBL=/etc/modprobe.d/raspi-blacklist.conf; [ -f ${MPBL} ] && sudo perl -p -i -e 's:^(blacklist (spi|i2c)-bcm2708):#$1:g' ${MPBL}

This line does the following:

* creates an environment variable named MPBL that contains the path to the file
* if the file exists it uses Perl to find the lines that contain the two module names
* if the lines were found, comments them out by prefixing them with a hash

While it might seem very fancy to run one-line commands that do stuff for you, it is much more convenient to do things the regular way, by editing a file in a normal text editor and seeing right away what you are doing.

After making the changes mentioned previously, you can restart your Raspberry Pi:

sudo reboot

### Verify that I2C can be accessed

After the device reboots, the first step is to check that the module is loaded.

sudo modprobe i2c-dev

If the module could not be loaded, this would return "modprobe: FATAL: Module i2c-dev not found”, otherwise it will return no message.

If all is well, you should be able to test the I2C connectivity by running the following command:

i2cdetect -y 1

If you have an older Raspberry Pi (before model B), the command would be instead:

i2cdetect -y 0

The makers of Raspberry Pi changed that 0 to a 1 when the model B was released. An easy way to remember is that any module with 256MB of RAM uses 0; the others use 1.

In either case, the output of the command would be similar to the following:

0 1 2 3 4 5 6 7 8 9 a b c d e f

00: -- -- -- -- -- -- -- -- -- -- -- -- --

10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

70: -- -- -- -- -- -- -- --

If you already have I2C devices connected, they might appear in the table, pointing to the address of your devices.

One interesting fact: if you have the PiGlow module plugged in, it will not show up in the i2cdetect output, but it will still work just fine.

If you are running a recent Raspberry Pi (3.18 kernel or higher) you will also need to update the /boot/config.txt file. Edit the /boot/config.txt file and uncomment or add the following lines:

dtparam=i2c1=on

dtparam=i2c\_arm=on

After you make these changes you need to reboot your device.

## Configuring GPIO

The GPIO pins can be used as both digital outputs and digital inputs. As digital outputs, you can write programs that turn a particular pin HIGH or LOW. Setting it HIGH sets it to 3.3V; setting it LOW sets it to 0V.

To drive an LED from one of these pins, you need a 1kΩ resistor in series with the LED as the GPIO pins can only manage a small amount of power.

If you use a pin as a digital input, then you can connect switches and simple sensors to a pin and then be able to check whether it is open or closed (i.e., activated or not). Following are some Adafruit projects that use just GPIO:

http://learn.adafruit.com/raspberry-pi-e-mail-notifier-using-leds

http://learn.adafruit.com/playing-sounds-and-using-buttons-with-raspberry-pi

http://learn.adafruit.com/basic-resistor-sensor-reading-on-raspberry-pi

To program the GPIO ports in Python, we need to install a very useful Python 2 library called RPi.GPIO (Raspberry Pi General Purpose Input Output). This module gives us a simple to use Python library that will let us control the GPIO pins. The installation process for this is the same whether you are using Raspbian or Occidentalis. In actual fact, some versions of Raspbian include this library, but these instructions will also have the effect of updating to the latest version, which is worth doing.

To install RPi.GPIO, you first need to install the Python Development toolkit that RPi.GPIO requires. To do this, enter the following command into LXTerminal:

sudo apt-get install python-dev

Then, to install Rpi.GPIO itself, type:

sudo apt-get install python-rpi.gpio

## Install PyGlow

To make the PyGlow module work with Python, we need to install the package that interacts with the I2C libraries and controls the device.

sudo pip install git+https://github.com/benleb/PyGlow.git

Now, you can create a small Python test script that would flash the lights blue for one second, red for two seconds, then green for 3 seconds (Listing 12-1).

Listing 12-1. Flashing the PiGlow LEDs

from PyGlow import PyGlow

from time import sleep

pyglow = PyGlow()

pyglow.all(0)

pyglow.color("blue", 100)

sleep(1)

pyglow.color("blue", 0)

pyglow.color("red", 100)

sleep(2)

pyglow.color("red", 0)

pyglow.color("green", 100)

sleep(3)

pyglow.color("green", 0)

Save the text in the Listing 12-1 in a file called flash.py, then run it in the SSH terminal:

python flash.py

This is all there is to it - simple, right?

Now imagine that the more complex control commands you will write will be nothing but encapsulated scripts such as the one in Listing 12-1, which take control of a given interface and device and effect some changes.

# Providing an API to control your device

We have seen so far that we have the ability to interact with attached devices on our Raspberry Pi. What we want now is to be able to interact with the devices from an external source, such as our iOS device.

There are two parts to this: the server on the Raspberry Pi that provides the API to control the attached devices (PiGlow in our case) and an iOS application that makes requests to this API.

Since we started writing Python code, it feels only right to do the API server also in Python. There are many Python frameworks that can be used to build an API: we need to use one that is lightweight and easy to customize. A good choice here is Flask. You will find a considerable number of online tutorials to help you get started with Flask, and as frameworks for API development, you will find out how easy it is to get started.

## Install Flask

This is a very simplified step-by-step tutorial to get started with Flask. We implement very basic features to allow us to control the device—following this model you can add security, scale the service to support multiple commands, or do whatever your project requires.

Before anything, we use Pip to install Flask:

sudo pip install flask

This will install the Flask framework and the basic modules used by the framework. The installation process is deceptively simple; it installs just like any other package.

### The hello world daemon

This is traditionally the first thing we would do to start experimenting with a new language.

Create a file named hello-flask.py with the contents shown in Listing 12-2. Normally you would configure a service to use port 80 since it is serving an HTTP response.

We use a higher number port for two reasons: to allow you to run a separate http server on port 80 if you need to, especially because the ports smaller than 1024 can only be used by programs that run with root privileges. It is an older limitation for security reasons, and it makes sure that applications running on certain well-known ports should be run only by the system.

Listing 12-2. The "Hello World!" script in Python using Flask

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return "Hello World!"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host='0.0.0.0', port=8080, debug=True)

When you start the daemon from the command line, it will tell us that it is running and show any incoming calls on the command line. This is what we get when we start it the first time. You will notice that the browser is trying to also get the favicon.ico file, a file that we do not have, because the daemon that we build in this example is not configured to serve static files directly. The favicon.ico is the image that can customize how your web site icon shows up to the left of the URL in the browser, and it will only show up once, then the browser will cache the state and will not try to fetch it again (Listing 12-3).

Listing 12-3. Running our Flask program

pi@raspberrypi:~$ python hello-flask.py

\* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)

\* Restarting with stat

10.0.1.25 - - [12/Oct/2015 05:26:18] "GET / HTTP/1.1" 200 -

10.0.1.25 - - [12/Oct/2015 05:26:18] "GET /favicon.ico HTTP/1.1" 404 -

The last two lines in Listing 12-3 are the result of loading the URL of the device, using the device IP (Internet Protocol) address:

http://10.0.1.128:8080/

In our example code we will not go to the length of setting up a discovery daemon or any fancy features, limiting ourselves to demonstrating functionality.

### Building a very simple listener daemon

With our knowledge accumulated with this test program, it is time to integrate a call that executes the commands we put together in the previous example. What we want to build is a daemon that informs us of the system time and offers a service to execute a simple command. Since we already spent some time writing a bit of code in the previous example, we will integrate that code in our listener. You can see the results in Listing 12-4.

You will notice that we have all the imports at the top, combining the needs of all the code written in this script.

Listing 12-4. The Listener Daemon on Raspberry Pi, written in Python and Flask

from flask import Flask

from PyGlow import PyGlow

from time import sleep

import datetime

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

now = datetime.datetime.now()

return now.strftime("%Y-%m-%d %H:%M")

@app.route("/blink")

def getData():

pyglow = PyGlow()

pyglow.all(0)

pyglow.color("blue", 100)

sleep(1)

pyglow.color("blue", 0)

pyglow.color("red", 100)

sleep(2)

pyglow.color("red", 0)

pyglow.color("green", 100)

sleep(3)

pyglow.color("green", 0)

return "OK"

@app.route("/blink/<color>")

def blinkColor(color):

pyglow = PyGlow()

pyglow.all(0)

pyglow.color(color, 100)

sleep(1)

pyglow.color(color, 0)

return "OK"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host='0.0.0.0', port=8080, debug=True)

With this we have our first very simple command handler daemon. Try now to bring up the base URL, and it will display the current date and time:

http://10.0.1.128:8080/

This Flask daemon is now taking a command to blink the lights on PiGlow, when somebody calls the /blink end point. When invoking the URL, you will see that our code is being executed and the LEDs will blink just as in our earlier example; then, when completed, the page will show "OK":

http://10.0.1.128:8080/blink

We also have a second service that allows us to blink a single color: for that, call the URLs:

http://10.0.1.128:8080/blink/green

<http://10.0.1.128:8080/blink/red>

The listener daemon shows the calls on the command line, as they are being made:

10.0.1.25 - - [12/Oct/2015 06:04:44] "GET /blink/green HTTP/1.1" 200 -

10.0.1.25 - - [12/Oct/2015 06:05:03] "GET /blink/red HTTP/1.1" 200 -

I am sure you will derive a lot of enjoyment from trying to add new commands and features.

# Setting up an iOS project for our app

We begin by creating an empty, single-page project. This chapter aims to show how to communicate with the Raspberry Pi API to the I2C interface we just created, not how to build an UI interface around it, so our application will be minimalistic, exposing just a few UI elements to trigger actions on the Raspberry Pi. We will be using a similar approach to the code in Chapter 5, so this will be familiar ground if you read that chapter.

Using this demo application, you will be able to trigger commands that switch the lights on and off on the PiGlow board on your Raspberry Pi.

## Allowing outgoing HTTP calls

You will run into this quite often: you rig your app to make an HTTP call and you see a stack trace like the following:  
Application Transport Security has blocked a cleartext HTTP (http://) resource load since it is insecure. Temporary exceptions can be configured via your app's Info.plist file.

Going to the Apple docs, we read about Application Transport Security:

App Transport Security (ATS) lets an app add a declaration to its Info.plist file that specifies the domains with which it needs secure communication. ATS prevents accidental disclosure, provides secure default behavior, and is easy to adopt. You should adopt ATS as soon as possible, regardless of whether you’re creating a new app or updating an existing one.

If you’re developing a new app, you should use HTTPS exclusively. If you have an existing app, you should use HTTPS as much as you can right now, and create a plan for migrating the rest of your app as soon as possible.

To handle this issue, you need to create an entry for "Allow Arbitrary Loads" set to "true" in your info.plist, as shown in Figure 12-2. XCode will autosuggest the name (App Transport Security Settings) the moment you start typing; it will also suggest the Dictionary type, and the first key-value pair (Allow Arbitrary Loads). You can see how this appears in the Figure 12-2.

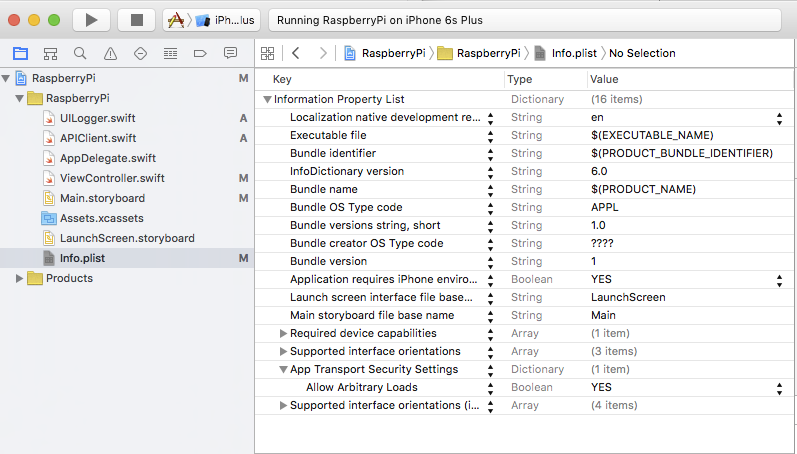


Figure 12-2. Setting the Allow Arbitrary Loads for the App Transport Security Settings

## 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 as optional (Listing 12-5).

Listing 12-5. The Header of the UIViewController Class

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!

In the viewDidLoad() function(Listing 12-6) 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 12-6. The viewDidLoad Override Function

override func viewDidLoad() {

super.viewDidLoad()

// Do any additional setup after loading the view, typically from a nib.

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 12-7):

Listing 12-7. The clickButton Function

@IBAction func clickButton() {

logger.logEvent("=== Blink All Lights ===")

api.blinkAllLights()

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

}

We can wire these button actions in the storyboard as shown in Figure 12-3.

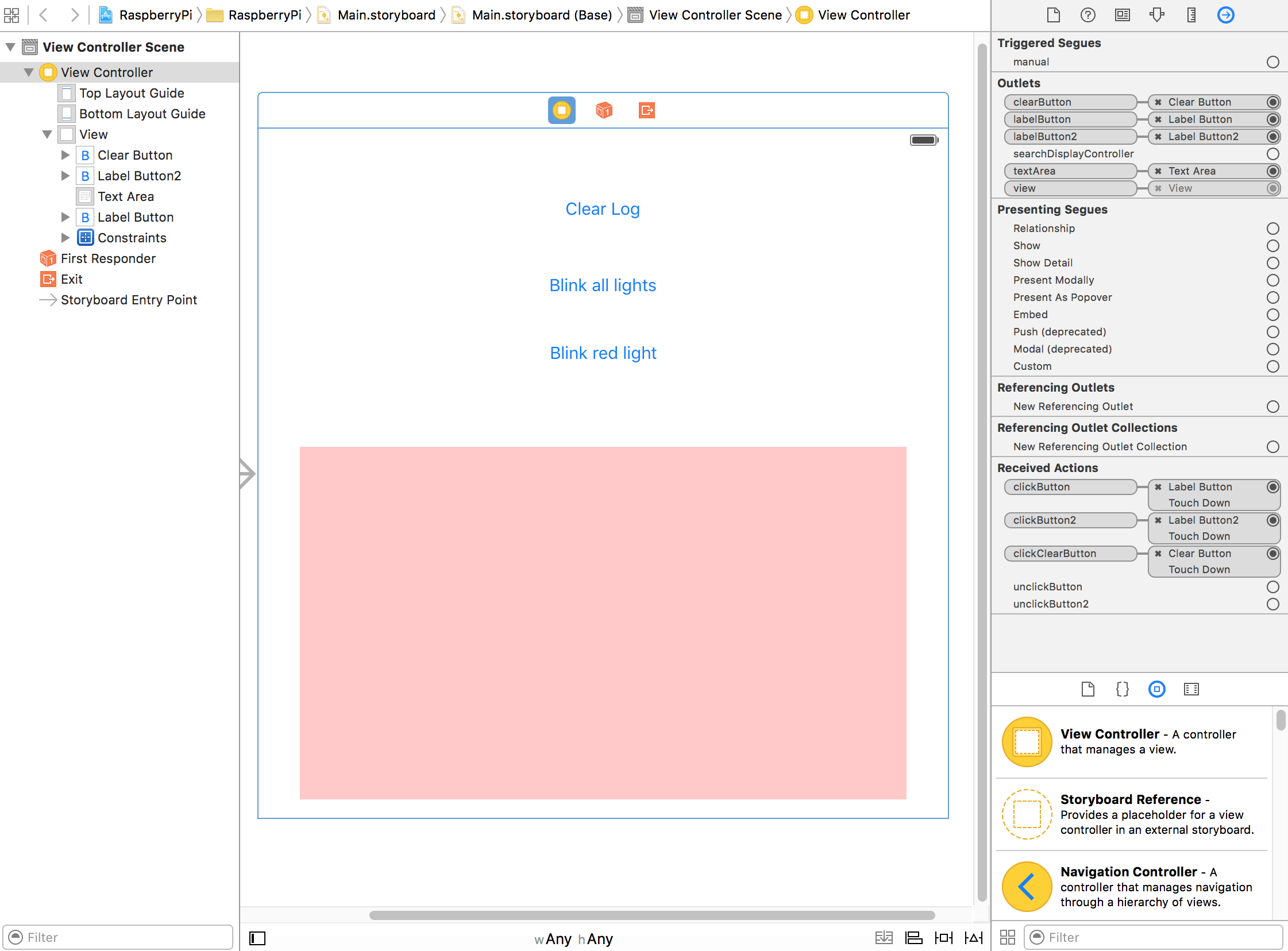


Figure 12-3. The storyboard

In Listing 12-8 we can see the entire ViewController.swift code that we will use to test the command sent to the Raspberry Pi API.

Listing 12-8. The Code for the ViewController.swift

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()

// Do any additional setup after loading the view, typically from a nib.

api = APIClient(parent: self)

logger = UILogger(out: textArea)

}

override func didReceiveMemoryWarning() {

super.didReceiveMemoryWarning()

// Dispose of any resources that can be recreated.

}

@IBAction func unclickButton() {

labelButton.setTitle("Blink All Lights", forState: UIControlState.Normal)

}

@IBAction func unclickButton2() {

labelButton2.setTitle("Blink Red Light", forState: UIControlState.Normal)

}

@IBAction func clickButton() {

logger.logEvent("=== Blink All Lights ===")

api.blinkAllLights()

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

}

@IBAction func clickButton2() {

logger.logEvent("=== Blink Red Light ===")

api.blinkLight("red")

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

}

@IBAction func clickClearButton() {

logger.set()

}

}

## 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.

When the main thread updates an UI element, there is no need for special treatment. However, since the calls that write to this log are running in a child thread, we make sure that updates to an UI element will be dispatched as async (Listing 12-9).

Listing 12-9. Dispatching an async Event

func set(text: String?="") {

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

self.textArea!.text = text

};

}

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. You can see the entire code of the UILogger.swift file in Listing 12-10.

Listing 12-10. The UILogger Library

import Foundation

import UIKit

class UILogger {

var textArea : UITextView!

required init(out: UITextView) {

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

self.textArea = out

};

self.set()

}

func set(text: String?="") {

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

self.textArea!.text = text

};

}

func logEvent(message: String) {

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

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

};

}

}

### The API client library

Now we will create now the APIClient.swift library. This library is for the functions that make async requests to the API. The header of the class contains the URL and other variables needed for the API functionality.

#### Defining the device URL

You can create a pop-up screen to enter this on the device, or you can write a discovery service using Bonjour to discover the device. To keep things simple, we hard-coded the IP address and the port of the device as a variable (Listing 12-11).

Listing 12-11. The Header of the APICLient Library

import Foundation

class APIClient {

var apiVersion: String!

var baseURL: String = "http://10.0.1.128:8080"

var viewController: ViewController!

required init (parent: ViewController!) {

viewController = parent

}

}

#### Creating a GET handler

A generic function to perform a GET from a service looks like the code in Listing 12.12. This code is part of the APIClient class, which we saved as the APIClient.swift file.

Listing 12-12. Generic Function to Perform a GET

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

let blockSelf = self

let 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. Since we do not know what data we will send to the API, 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 shown in Listing 12-13.

Listing 12-13. Implementing a GET Request Handler

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

// do something with data here

}

#### Creating a POST handler

Just like the GET request, the POST request can have the same structure seen in Listing 12-14. This, too, is part of the APIClient class (the APIClient.swift file).

Listing 12-14. Implementing a POST Request Handler

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

let blockSelf = self

let 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.

#### Defining the verbs and services

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 allows us to define which HTTP verbs are supported by the API client (Listing 12-15). This too, is part of the APIClient class (the APIClient.swift file).

Listing 12-15. The APIMethod Enum

enum APIMethod {

case GET, POST

func toString() -> String {

var method: String!

switch self {

case .GET:

method = "GET"

case .POST:

method = "POST"

}

return method

}

}

We provide the hasBody()function 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 PUT and POST use it as JSON (JavaScript Object Notation).

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 Listing 12-16 shows the basic format.

Listing 12-16. The APIService Enum

enum APIService {

case BLINK

func toString() -> String {

var service: String!

switch self {

case .BLINK:

service = "blink"

}

return service

}

}

#### Adding an extension to the String type

We add in the same APIClient.swift file an extension to the string type to add a simple escaping method for the URL params we might want to pass to a call (Listing 12-17).

Listing 12-17. The Extension to the String Object

extension String {

func escapeUrl() -> String {

let source: NSString = NSString(string: self)

let chars = "abcdefghijklmnopqrstuvwxyz"

let okChars = chars + chars.uppercaseString + "0123456789.~\_-"

let customAllowedSet = NSCharacterSet(charactersInString: okChars)

return source.stringByAddingPercentEncodingWithAllowedCharacters(customAllowedSet)!

}

}

#### The apiRequest() function

Next to be defined is the apiRequest() function(Listing 12-18). This function will make the actual API request, and that includes handling the 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. This is also part of the APIClient class.

Listing 12-18. The apiRequest Function

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 INFO and BLINK: the API overloads them with a variable list of params, so in essence your calls will need to provide the larger APIService, then provide via the urlSuffix the URL path extension to point to the right resource. This will be explained in more detail later.

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

1. Compose the base URL of the service
2. Add the URL suffix if it was specified
3. Serialize and append to the URL the input params
4. Make the API request as an async call

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

1. De-serialize the JSON response, if JSON was found

2. Call the callback function

To **compose the base URL of the service** we use the code shown in Listing 12-19.

Listing 12-19. Composing 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()

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 12-20 to serialize the input data.

Listing 12-20. Serializing JSON Data

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")

**To handle the composition of the URL**, we create the asURLString() function. This function is located in the APIClient class and takes a dictionary of input params and creates a URL-encoded string, with the parameters sorted alphabetically (Listing 12-21).

Listing 12-21. The asURLString Function

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

var params: [String] = []

for (key, value) in inputData {

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

}

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

return params.joinWithSeparator("&")

}

****The URL suffix**** needs to be made part of the URL— - we have 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. You can find the following example (Listing 12-22) in the postData() function in the APIClient class:

Listing 12-22. Composing a URL

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

if urlSuffix?.count > 0 {

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

}

****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 the 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 we invoked apiRequest(). 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, and dictionaries, and NSDictionary supports by default AnyObject types. The NSJSONReadingOptions.MutableContainers specifies that arrays and dictionaries be created as mutable objects. We can see this in Listing 12-23. This code is located in the postData() function in the APIClient class.

Listing 12-23. Parsing a JSON Response

var jsonResult: NSDictionary?

if urlResponse != nil {

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

if data != nil {

do {

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

} catch {

// we expect an “OK” from the API, not JSON, so it’s OK if we don’t do anything here

}

}

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 textarea 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. We can see the results in Listing 12-24. This code is located in the postData() function in the APIClient class.

Listing 12-24. Handling a REST Call

let logger: UILogger = viewController.logger

let session = NSURLSession.sharedSession()

let task = session.dataTaskWithRequest(request) { (data : NSData?, urlResponse : NSURLResponse?, 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 {

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

if data != nil {

do {

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

} catch {

// we expect an “OK” from the API, not JSON, so it’s OK if we don’t do anything here

// print("json error: \(error)")

}

}

logger.logEvent("RESPONSE RAW: " + (rData.isEmpty ? "No Data" : rData) )

print("RESPONSE RAW: \(rData)")

}

else {

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

}

callback(responseJson: jsonResult, responseError: error)

}

task.resume()

Displaying pretty-formatted JSON can be useful in other places too, so we extracted the code in Listing 12-25 in the prettyJSON() function. This code is located in the postData() function in the APIClient class.

Listing 12-25. Displaying Pretty-Formatted JSON Responses

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

var pretty: String!

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

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

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

}

}

return pretty

}

Listing 12-26 shows the entire code we have so far for the APIClient library.

Listing 12-26. The APIClient.swift Library

import Foundation

class APIClient {

var apiVersion: String!

var baseURL: String = "http://10.0.1.128:8080"

var viewController: ViewController!

required init (parent: ViewController!) {

viewController = parent

}

func blinkAllLights () {

// GET /blink

getData(APIService.BLINK)

}

func blinkLight(color: String) {

// GET /blink/red

getData(APIService.BLINK, id: color)

}

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

let blockSelf = self

let 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]!=[:]) {

let blockSelf = self

let 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

}

let 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("/")

}

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

request.URL = NSURL(string: serviceURL)

if !inputData.isEmpty {

serviceURL += "?" + asURLString(inputData)

request.URL = NSURL(string: serviceURL)

}

//now make the request

let logger: UILogger = viewController.logger

let session = NSURLSession.sharedSession()

let task = session.dataTaskWithRequest(request) { (data : NSData?, urlResponse : NSURLResponse?, 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 {

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

if data != nil {

do {

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

} catch {

// we expect an “OK” from the API, not JSON, so it’s OK if we don’t do anything here

// print("json error: \(error)")

}

}

logger.logEvent("RESPONSE RAW: " + (rData.isEmpty ? "No Data" : rData) )

print("RESPONSE RAW: \(rData)")

}

else {

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

}

callback(responseJson: jsonResult, responseError: error)

}

task.resume()

}

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

var params: [String] = []

for (key, value) in inputData {

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

}

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

return params.joinWithSeparator("&")

}

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

var pretty: String!

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

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

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

}

}

return pretty

}

}

extension String {

func escapeUrl() -> String {

let source: NSString = NSString(string: self)

let chars = "abcdefghijklmnopqrstuvwxyz"

let okChars = chars + chars.uppercaseString + "0123456789.~\_-"

let customAllowedSet = NSCharacterSet(charactersInString: okChars)

return source.stringByAddingPercentEncodingWithAllowedCharacters(customAllowedSet)!

}

}

enum APIService {

case BLINK

func toString() -> String {

var service: String!

switch self {

case .BLINK:

service = "blink"

}

return service

}

}

enum APIMethod {

case GET, POST

func toString() -> String {

var method: String!

switch self {

case .GET:

method = "GET"

case .POST:

method = "POST"

}

return method

}

}

You should now be able to send commands to the device, and you will see the results in the text area as shown in Figure 12-4.

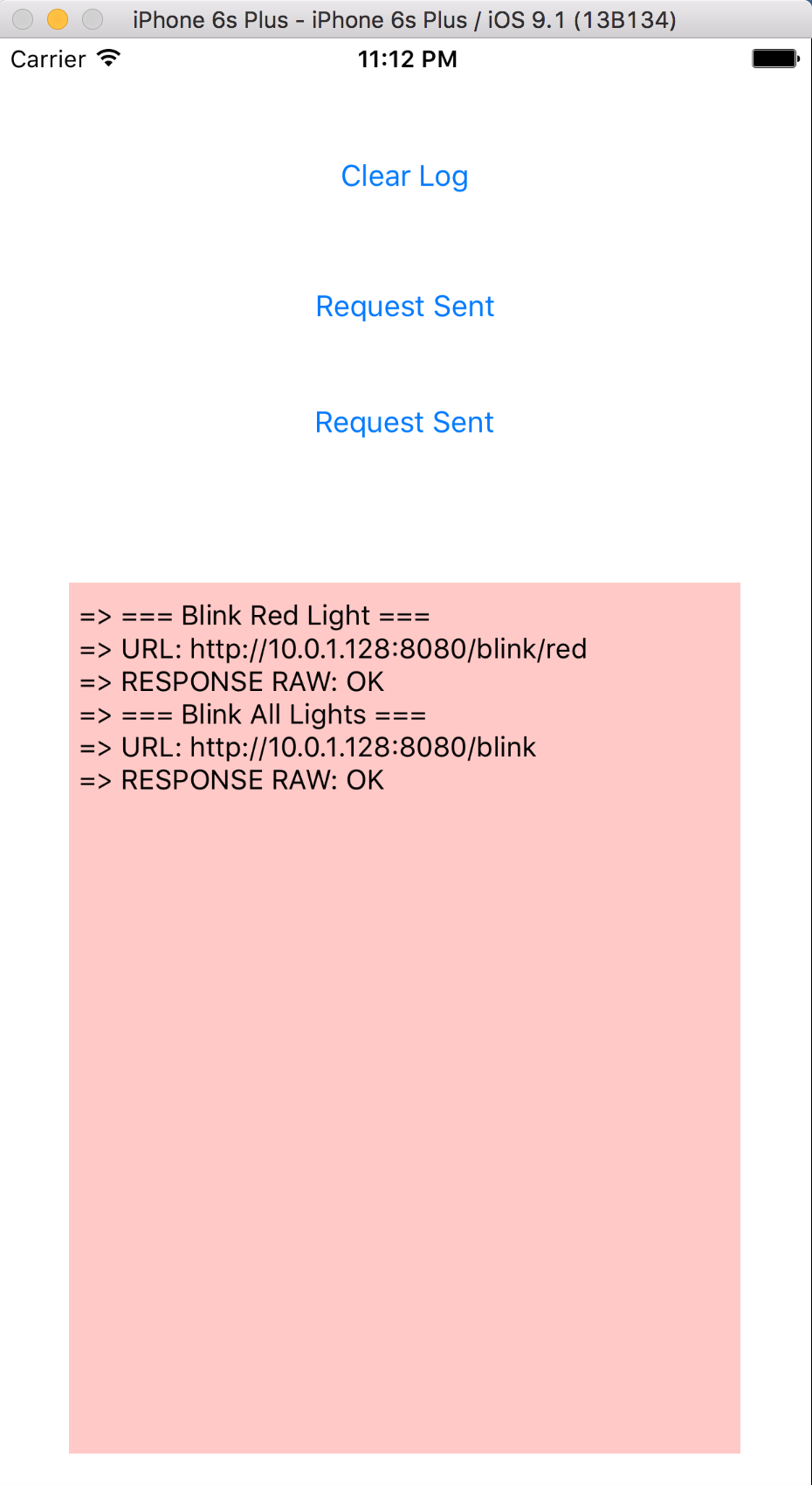


Figure 12-4. The request/response view of the application

# Summary

In this chapter, you learned to set up a basic script that interacts with resources on a Raspberry Pi, the service necessary to run a listener that takes remote commands, and how to write a very basic iOS application that makes an HTTP request to interact with the device via the very simple API you created.