Francesco Azzola

Android Things Projects

Effeciently build IoT projects with Android Things

Packt

Android Things Projects

Effeciently build IoT projects with Android Things

Francesco Azzola

BIRMINGHAM - MUMBAI

< html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-html40/loose.dtd">

Android Things Projects

Copyright © 2017 Packt Publishing

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

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

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

First published: June 2017

Production reference: 1290617

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78728-924-6

www.packtpub.com

Credits

Author Francesco Azzola	**Copy Editor** Â Safis Editing
ReviewersÂ Â Ali Utku Selen Raimon RÃ fols Montane	**Project Coordinator** Kinjal BariÂ
Commissioning Editor Â Vijin Boricha	**Proofreader** Â Safis Editing
Acquisition Editor Â Namrata Patil	**Indexer** Â Mariammal Chettiyar

Content Development Editor Â Mamata Walkar	**Graphics** Â Kirk D'Penha
Technical Editor Â Varsha Shivhare	**Production Coordinator** Melwyn Dsa

About the Author

Francesco Azzola is an electronic engineer with over 15 years of experience in computer programming and JEE architecture. He is a Sun Certified Enterprise Architect (SCEA), SCWCD, and SCJP. He is an Android and IoT enthusiast who loves creating IoT projects using Arduino, Raspberry Pi, Android, and other platforms.

He is interested in the convergence of IoT and mobile applications. Previously, he worked in the mobile development field for several years. He has created a blog called *Surviving with Android*, where he shares posts on coding in Android and IoT projects.

About the Reviewers

Ali Utku Selen is a system engineer at Sony Mobile, who has been working on flagship Android devices for more than five years. He started programming at age 11, and since then he has developed a great interest in software development. He holds an MSc degree from the Computer Engineering Department of Dokuz Eylul University.

Raimon RÃ folsÂ MontaneÂ has been developing for mobile devices since 2004. He has experience in developing on several technologies specializing in UI, build systems, and client-server communications. He is currently working as a Engineering Manager at AXA Group Solutions in Barcelona, although has been working in the past for Imagination Technologies near London and Service2Media in the Netherlands. In his spare time, he enjoys programming, photography, and giving talks at mobile conferences about Android performance optimization and Android custom views.

www.PacktPub.com

For support files and downloads related to your book, please visitÂ www.PacktPub.com.

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

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

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.com/dp/1787289249.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

- Preface
  - What this book covers
  - What you need for this book
  - Who this book is for
  - Conventions
  - Reader feedback
  - Customer support
    - Downloading the example code
    - Downloading the color images of this book
    - Errata
    - Piracy
    - Questions

1. Getting Started with Android Things
   - Internet of Things overview
   - IoT components
   - Android Things overview
   - Things support library
   - Android Things board compatibility
   - How to install Android Things on Raspberry
     - How to install Android Things using Windows
     - How to install Android Things using OS X
     - Testing the installation
   - How to install Android Things on Intel Edison
   - Configuring the WiFi
   - Creating the first Android Things project
     - Cloning the template project
     - Create the project manually
   - Differences between Android and Android Things
   - Create your first Android Things app

Summary

2. Creating an Alarm System Using Android Things
    Alarm system project description
        PIR sensor

        Project schematic

        How to use GPIO pins

        Reading from the GPIO pin

        How to add a listener to GPIO
            Declare the event to listen to

            Implementing the callback class

    How to close the connection with a GPIO pin

    Handle different boards in Android Things
        Android Things board pinout

        How to identify the board

    How to implement the notification system
        Configuring firebase

        Add the notification feature to the Android Things app

    Android companion app

    Summary

3. How to Make an Environmental Monitoring System
    Environmental monitoring system project overview
        Project components

        Project schematic

    How to read data from sensors

    Handling sensors using the Android sensor framework
        Implementing the sensor callback

        How to handle dynamic sensors

    Putting it all together - acquiring data

    How to close the sensor connection

    How to control GPIO pins
        Initialize the GPIO pin

    Diving into I2C protocol
        I2C protocol overview

    How to implement a custom sensor driver

Low-level sensor driver

Summary

4. Integrate Android Things with IoT Cloud Platforms
IoT cloud architecture
An IoT cloud platform overview

IoT cloud architecture overview

Streaming data to the IoT cloud platform
How to configure Artik Cloud

Artik client description

How to implement the Android Things Artik client
Implement a StringRequest with Volley

Implement a custom HTTP header

Send the data using a custom body request

Sending data from the Android Things app

Creating a dashboard
Data logging

Adding voice capabilities to Android Things
Configure Temboo choreo

Integrate Temboo in the Android Things app

Summary

5. Create a Smart System to Control Ambient Light
Ambient light control system description
Project components

Project architecture

Building the Arduino project
How Arduino exposes the services

Implementing the Android Things app
How to develop an Android Things app UI

Attaching the layout to the Activity

Handling UI events

Invoking the Arduino services

How to implement a web interface
Implementing a simple HTTP web server

Creating the HTML page with the UI

Embedding the HTTP Server into the Android Things app

Summary

6. Remote Weather Station
    Remote weather station project description
        Project components

    The M2M architecture and the MQTT protocol
        MQTT protocol overview
            MQTT message details
                Security and QoS

        Using MQTT in our remote weather station
            Implementing the MQTT publisher
                Connecting to MQTT and sending data

            Implementing the MQTT subscriber using Android Things

        Implementing the Android Things Activity

        Displaying the information using OLED display

        Connect the OLED display to Android Things board

    Installing the MQTT server
        Installing the MQTT broker
            Configuring the MQTT broker

    Summary

7. Build a Spying Eye
    Spying eye Android Things project overview
        Project components

        Pulse Width Modulation overview

        How to use PWM with Android Things

    Implementing the spying eye project in Android Things
        Controlling a servomotor in Android Things

        Using a camera in Android Things
            Getting ready to use the camera

        Assembling the app

    Summary

8. Android with Android Things
    Architecture to connect Android and Android Things

    How to control a LED strip using an Android app
        Connecting the Android app to Android Things

    How to develop an Android app that retrieves data from Android Things
        How to implement a Bluetooth connection

Creating the Android app

Implementing the Bluetooth server in Android Things

Summary

Preface

Android Things is the new OS developed by Google for building professional IoT projects using Android. Throughout the course of this book, you will gain deep knowledge of Android Things and get ready for the next technological revolution. You will learn how to create real-life IoT projects covering all the aspects of Android Things.

What this book covers

Chapter 1, *Getting Started with Android Things*, introduces IoT and explains why it has such huge impact on everyday life. This chapter also introduces Android Things and explains how to use it in your first IoT project.

Chapter 2, *Creating an Alarm System Using Android Things*, shows how to use two-state sensors (or binary devices) in Android Things. This chapter also covers creating an alarm system that detects motion and sends a notification to a user's smartphone.

Chapter 3, *How to Make an Environmental Monitoring System*, shows how to connect sensors to Android Things and how to read data using the I2C bus. These concepts are applied to an IoT project that monitors the environmental parameters and uses an RGB LED to visualize it.

Chapter 4, *Integrate Android Things with IoT Cloud Platforms*, covers how to use Android Things in an IoT cloud architecture. This chapter describes how to stream real-time data from sensors to IoT cloud platforms.

Chapter 5, *Create a Smart System to Control Ambient Light*, demonstrates how to use a simple integration pattern to integrate Android Things with Arduino using the HTTP protocol.

Chapter 6, *Remote Weather Station*, covers how to use Android Things in **Machine to Machine** (**M2M**) architecture. In this chapter, we will build a remote weather station that monitors temperature, humidity, pressure, and light, and sends data using the MQTT protocol.

Chapter 7, *Build a Spying Eye*, shows how to develop an Android Things app that controls servo motors using **Pulse Width Modulation** (**PWM**) and how to use the camera with Android Things.

Chapter 8, *Android with Android Things*, covers how to develop Android companion apps that interact with Android things.

What you need for this book

To build the examples in this book, you need to have Windows OS or Mac OS X. Moreover, in order to develop, compile, and install the Android Things app, you have to install Android Studio as specified during the chapters.

Who this book is for

This book is for Android enthusiasts, hobbyists, IoT experts, and Android developers who want to gain deep knowledge of Android Things. The main focus is on implementing IoT projects using Android Things. This book also covers Android Things API and how to use them in IoT. The reader will use sensors, resistors, capacitors, and IoT cloud platforms.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
adb shell am startservice
-n com.google.wifisetup/.WifiSetupService
-a WifiSetupService.Connect
-e ssid <Your_WIFI_SSID>
-e passphrase <WIFI_password>
```

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

```
sudo dd bs=1m if=path_of_your_image.img of=/dev/rdiskn
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in text like this:

"When the board is connected to your PC or Mac, it appears in the Platform Flash Tool Light."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book what you liked or disliked. Reader feedback is important to us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

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

Downloading the example code

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

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Android-Things-Projects. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/AndroidThingsProjects_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Getting Started with Android Things

Recently, Google released its first operating system built for the Internet of Things, called Android Things. During this book, you will learn how to build IoT projects using this OS with compatible development boards and peripherals such as sensors, LEDs, servo, and so on.

In this chapter, at the beginning, we will give an overview of Android Things, covering what it is and how it differs from Android. You will learn how to reuse your Android knowledge in Android Things projects. After this, you will learn how to install Android Things on different boards such as Raspberry Pi 3 and Intel Edison with an Arduino breakout kit. This will help you to familiarize yourself with development boards while we set up our development environment. Once you are comfortable with it, we will move to creating the first Android Things project and you will learn how to use simple peripherals such as LEDs and buttons (or switches). In more detail, we will explore how to convert an Android project into an Android Things project. Moreover, you will have an overview of the most import Android Things API and how to use it in a real IoT project.

The main topics covered in this chapter are:

- Internet of things overview Android Things layer structure
- How to install Android Things on Raspberry Pi 3
- How to install Android Things on Intel Edison with Arduino breakout kit
- How to create an Android Things project

Internet of Things overview

Internet of Things, or briefly **IoT,** is one of the most promising trends in technology. According to many analysts, Internet of things can be the most disruptive technology in the upcoming decade. It will have a huge impact on our lives and it promises to modify our habits. IoT is and will be in the future a pervasive technology that will span its effects across many sectors:

- Industry
- Healthcare
- Transportation
- Manufacturing
- Agriculture
- Retail
- Smart cities

All these areas will benefit from using IoT. Before diving into IoT projects, it is important to know what IoT means. There are several definitions about the Internet of things, addressing different aspects and considering different areas of application. Anyway, it is important to underline that the IoT is much more than a network of smartphones, tablets, and PCs connected to each other. Briefly, IoT is an ecosystem where objects are interconnected and, at the same time, they connect to the internet. The Internet of things includes every object that can potentially connect to the internet and exchange data and information. These objects are always connected anytime, anywhere, and they exchange data.

The concept of connected objects is not new and over the years it has been developed. The level of circuit miniaturization and the increasing power of CPU with a lower consumption makes it possible to imagine a future where there are millions of "things" that talk to each other.

The first time that the Internet of things was officially recognized was in 2005. The **International Communication Union** (**ITU**) in a report titled *The Internet of things* (https://www.itu.int/osg/spu/publications/internetofthings/InternetofThings_summary.pdf), gave the first definition:

"A new dimension has been added to the world of information and communication technologies (ICTs): from anytime, any place connectivity for anyone, we will now have connectivity for anything ... . Connections will multiply and create an entirely new dynamic network of networks - an Internet of Things "

In other words, the IoT is a network of smart objects (or things) that can receive and send data and we can control it remotely.

IoT components

There are several elements that contribute to creating the IoT ecosystem and it is important to understand the role they play in order to have a clear picture about IoT. This will be useful to better understand the projects we will build using Android Things. The basic brick of IoT is a smart object. It is a device that connects to the internet and it is capable of exchanging data. It can be a simple sensor that measures a quantity such as pressure, temperature, and so on, or a complex system. Extending this concept, our oven, our coffee machine, and even our washing machine are all examples of smart objects once they connect to the internet. All of these smart objects contribute to developing the internet of things network. Anyway, it's not only household appliances that are examples of smart objects, but also cars, buildings, actuators, and so on. We can reference these objects, when connected, using a unique identifier and start talking to them.

At the low level, these devices exchange data using a network layer. The most important and known protocols at the base of Internet of things are:

- Wi-Fi
- Bluetooth
- Zigbee
- Cellular network
- NB-IoT
- LoRA

From an application point of view, there are several application protocols widely used in the internet of things. Some protocols derive from different contexts (such as the web); others are IoT-specific. To name a few of them, we can remember:

- HTTP
- MQTT
- CoAP
- AMQP
- Rest
- XMPP
- Stomp

By now, they could be just names or empty boxes, but throughout this book we will explore how to use these protocols with Android Things.

Prototyping boards play an important role in the Internet of things and they help to develop the number of connected objects. Using prototyping boards, we can experiment with IoT projects and in this book, we will explore how to build and test IoT projects using boards compatible with Android Things. As you may already know, there are several prototyping boards available on the market, each one having specific features. Just to name a few of them, we can list:

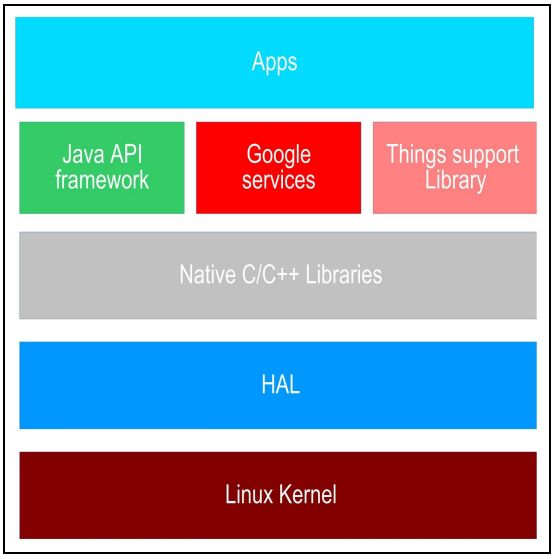
- Arduino (in different flavors)
- Raspberry Pi (in different flavors)

- Intel Edison
- ESP8266
- NXP

We will focus our attention on Raspberry Pi 3 and Intel Edison because Android Things officially supports these two boards. During the books, we will also use other development boards so that you can understand how to integrate them.

Android Things overview

Android Things is the new operating system developed by Google to build IoT projects. This helps you to develop professional applications using trusted platforms and Android. Yes Android, because Android Things is a modified version of Android and we can reuse our Android knowledge to implement smart Internet of things projects. This OS has great potential because Android developers can smoothly move to IoT and start developing and building projects in a few days. Before diving into Android Things, it is important to have an overview. Android Things OS has the layer structure shown in the following diagram:



Source: https://developer.android.com/things/sdk/index.html

This structure is slightly different from Android OS because it is much more compact so that apps for Android Things have fewer layers beneath and they are closer to drivers and peripherals than **normal** Android apps. Even if Android Things derives from Android, there are some APIs available in Android not supported in Android Things. We will now briefly describe the similarities and the differences.

Let us start with the **content providers**, widely used in Android, and not present in Android Things SDK. Therefore, we should pay attention when we develop an Android Things app. To have more information about these content providers not supported, please refer to the Official Android Things website at https://developer.android.com/things/sdk/index.html.

Moreover, like a normal Android app, an Android Things app can have a **User Interface** (**UI**), even if this is optional, and it depends on the type of application we are developing. A user can interact with the UI to trigger events as they happen in an Android app. From this point of view, as we will see later, the developing process of a UI is the same as used in Android. This is an interesting feature because we can develop an IoT UI easily and fast, re- using our Android knowledge.

It is worth noting that Android Things fits perfectly in the Google services. Almost all cloud services implemented by Google are available in Android Things with some exceptions. Android Things does not support Google services strictly connected to the mobile world and those that require user input or authentication. Do not forget that user

interface for an Android Things app is optional. To have a detailed list of Google services available in Android Things refer to the official page at https://developer.android.com/things/sdk/index.html.

An important Android aspect is the permission management. An Android app runs in a sandbox with limited access to the resources. When an app needs to access a specific resource outside the sandbox it has to request permission. In an Android app, this happens in the `Manifest.xml` file. This is still true in Android Things and all the permissions requested by the app are granted at *installation time*. Android 6 (API level 23) has introduced a new way to request a permission. An app can request a permission not only at installation time (using the `Manifest.xml` file), but at run-time too. Android Things does not support this new feature, so we have to request all the permissions in the Manifest file.

The last thing to notice is the notifications. As we will see later, Android Things UI does not support the notification status bar, so we cannot trigger notifications from our Android Things apps.

To make things simpler, you should remember that all the services related to the user interfa require a user interface to accomplish the task are not guaranteed to work in Android Things

Things support library

Things support library is the new library developed by Google to handle the communication with peripherals and drivers. This is a completely new library not present in the Android SDK and this library is one of the most important features. It exposes a set of Java Interface and classes (APIs) that we can use to connect and exchange data with external devices such as sensors, actuators, and so on. This library hides the inner communication details, supporting several industry standard protocols such as:

- GPIO
- I2C
- PWM
- SPI
- UART

During the book, we will discover how to use this library to connect to several devices.

Moreover, this library exposes a set of APIs to create and register new device drivers called **user drivers**. These drivers are custom components deployed with the Android Things app that extends the Android Things framework. In other words, they are custom libraries that enable an app to communicate with other device types not supported by Android Things natively.

This book will guide you, step by step, to learn how to build real-life projects using Android. You will explore the new Android Things APIs and how to use them. In the next sections, you will learn how to install Android Things on Raspberry Pi 3 and Intel Edison.

Android Things board compatibility

Android Things is a new operating system specifically built for IoT. At the time of writing, Android Things supported four different development boards:

- Raspberry Pi 3 Model B
- Intel Edison
- NXP Pico i.MX6UL
- Intel Joule 570x

In the near future, more boards will be added to the list. Google has already announced that it will support this new board **NXP Argon i.MX6UL**.

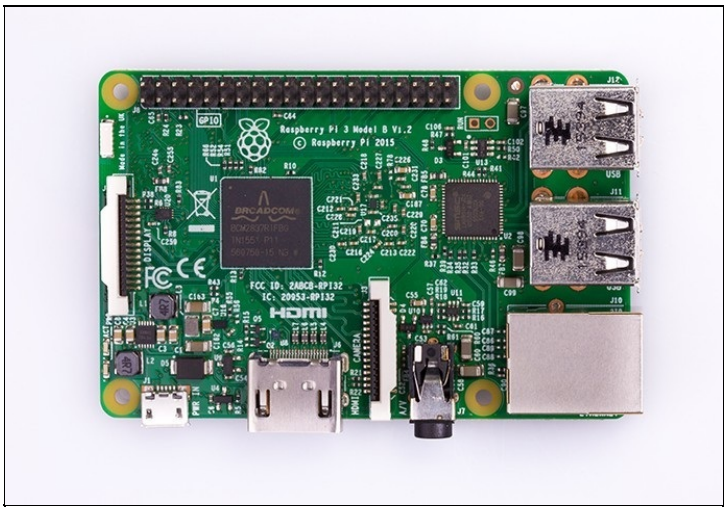
The book will focus on using the first two boards: Raspberry Pi 3 and Intel Edison. Anyway, you can develop and test all the book's projects on the other boards too. This is the power of Android Things: it abstracts the underlying hardware providing, a common way to interact with peripherals and devices. The paradigm that made Java famous, **Write Once and Run Anywhere** (**WORA**), applies to Android Things too. This is a winning feature of Android Things because we can develop an Android Things app without worrying about the underlying board. Anyway, when we develop an IoT app there are some minor aspects we should consider so that our app will be portable to other compatible boards.

How to install Android Things on Raspberry

Raspberry Pi 3 is the latest board developed by Raspberry. It is an upgrade of Raspberry Pi 2 Model B and like its predecessor it has some great features:

- Quad-core ARMv8 Cpu at 1.2Ghz
- Wireless Lan 802.11n
- Bluetooth 4.0

The following image shows a Raspberry Pi 3 Model B:



Source: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

In this section, you will learn how to install Android Things on Raspberry Pi 3 using a Windows PC or a macOS.

Before starting the installation process you must have:

- Raspberry Pi 3 Model B
- At least an 8Gb SD card
- A USB cable to connect Raspberry to your PC
- An HDMI cable to connect Raspberry to a tv/monitor (optional)

If you do not have an HDMI cable you can use a screen mirroring tool. This is useful to know the result of the installation process and when we will develop the Android Things UIs. The installation steps are different if you are using Windows, OS X, or Linux.

How to install Android Things using Windows

At the beginning we will cover how to install Android Things on Raspberry Pi 3 using a Windows PC:

1. Download the Android Things image from this link: https://developer.android.com/things/preview/download.html. Select the right image; in this case, you have to choose the Raspberry Pi image.
2. Accept the license and wait until the download is completed.
3. Once the download is complete, extract the ZIP file.
4. To install the image on the SD card, there is a great application called **Win32 Disk Imager** that works perfectly. It is free and you can download it from SourceForge at: https://sourceforge.net/projects/win32diskimager/. At the time of writing, the application version is 0.9.5.
5. After you have downloaded it, you have to run the installation executable as Administrator. Now you are ready to burn the image into the SD card.
6. Insert the SD card into your PC.
7. Select the image you have unzipped in step 3 and be sure to select the right disk name (your SD). At the end, click on **Write**.

You are done! The image is installed on the SD card and we can now start Raspberry Pi.

How to install Android Things using OS X

If you have a Mac OS X, the steps to install Android Things are slightly different. There are several options to flash this OS to the SD card; you will learn the fastest and easiest one.

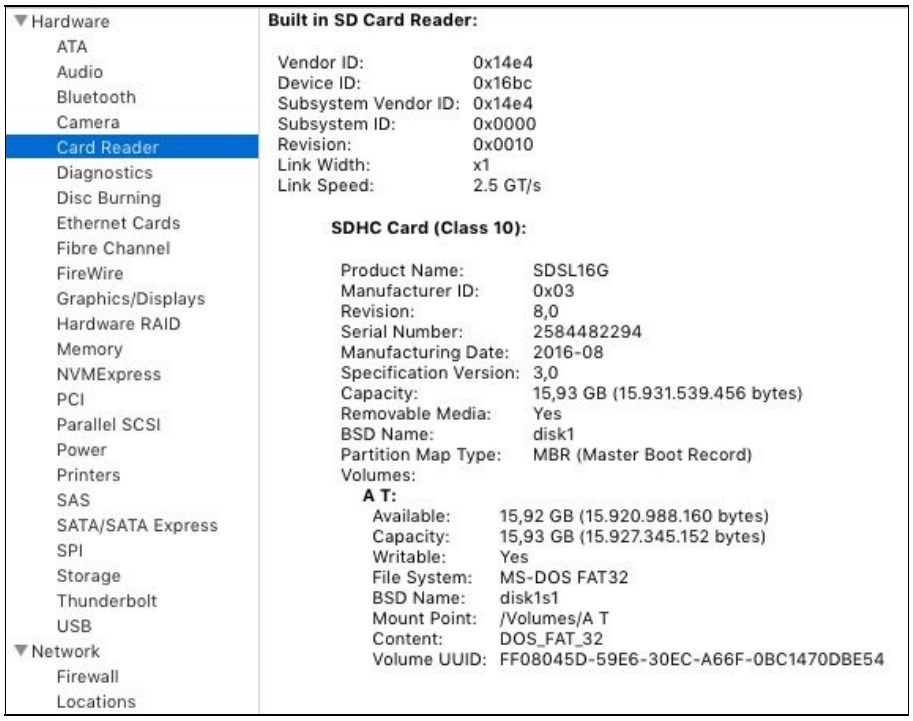
These are the steps to follow:

1. Format your SD card using FAT32. Insert your SD card into your Mac and run `Disk Utility`. You should see something like this:

2. Download the Android Things OS image using this link: https://developer.android.com/things/preview/download.html.
3. Unzip the file you have downloaded.
4. Insert the SD card into your Mac.
5. Now it is time to copy the image to the SD card. Open a terminal window and write the following:

```
sudo dd bs=1m if=path_of_your_image.img of=/dev/rdiskn
```

Where the `path_to_your_image` is the path to the file with the `img` extension you downloaded at step 2. In order to find out the `rdiskn` you have to select Preferences and then System Report. The result is shown in the following screenshot:



The BSD name is the disk name we are looking for. In this case, we have to write the following:

```
sudo dd bs=1m if=path_of_your_image.img of=/dev/disk1
```

That's all. You have to wait until the image is copied into the SD card. Do not forget that the copying process could take a while. So be patient!

Testing the installation

Once we have flashed the Android Things image into the SD card, we can remove it from the PC or Mac and insert it into the board:

1. Connect Raspberry Pi to a video using the HDMI.
2. Connect Raspberry Pi to your network using the LAN connection.
3. Connect Raspberry Pi to your Mac/PC using a USB cable.

Wait until Android Things completes the boot phase. At the end, you should see the following:

Now your development board is ready and we can start developing our first Android Things project. To confirm that your Android Things is up and running, you can execute from the command line the following command:

```
adb devices
```

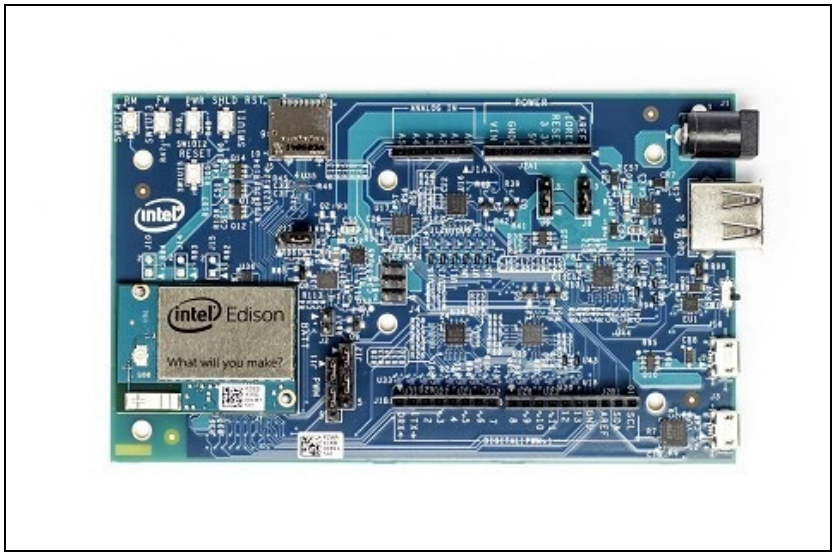
You should see, in the list, at least one Android device with an IP address. Congratulations; you have just installed and tested your Android Things OS. By now you should see the Android Things default screen because we did not install an app on the system.

How to install Android Things on Intel Edison

Intel Edison is a prototyping board developed by Intel with interesting features. It is a Raspberry Pi 3 alternative and it is powerful. The main specifications for this board are:

- Intel Dual-core Atom at 500MHz
- 1 Gb DDR3 Ram and 4 Gb eMMC flash
- Compatible with Arduino (using an Arduino breakout Kit)
- Bluetooth and WiFi

Intel Edison with Arduino Kit is shown in the following image:



source: https://www.arduino.cc/en/uploads/ArduinoCertified/Intel_Edison_Kit_Front.jpg

In this book, we will use Intel Edison and Arduino breakout kit to develop our projects. Anyway, you can apply all the topics covered here to other Intel development boards compatible with Android Things. Before starting to flash the image into the Intel board, be sure you have installed the following on your system:

- SDK Platform tools 25.0.3 or later

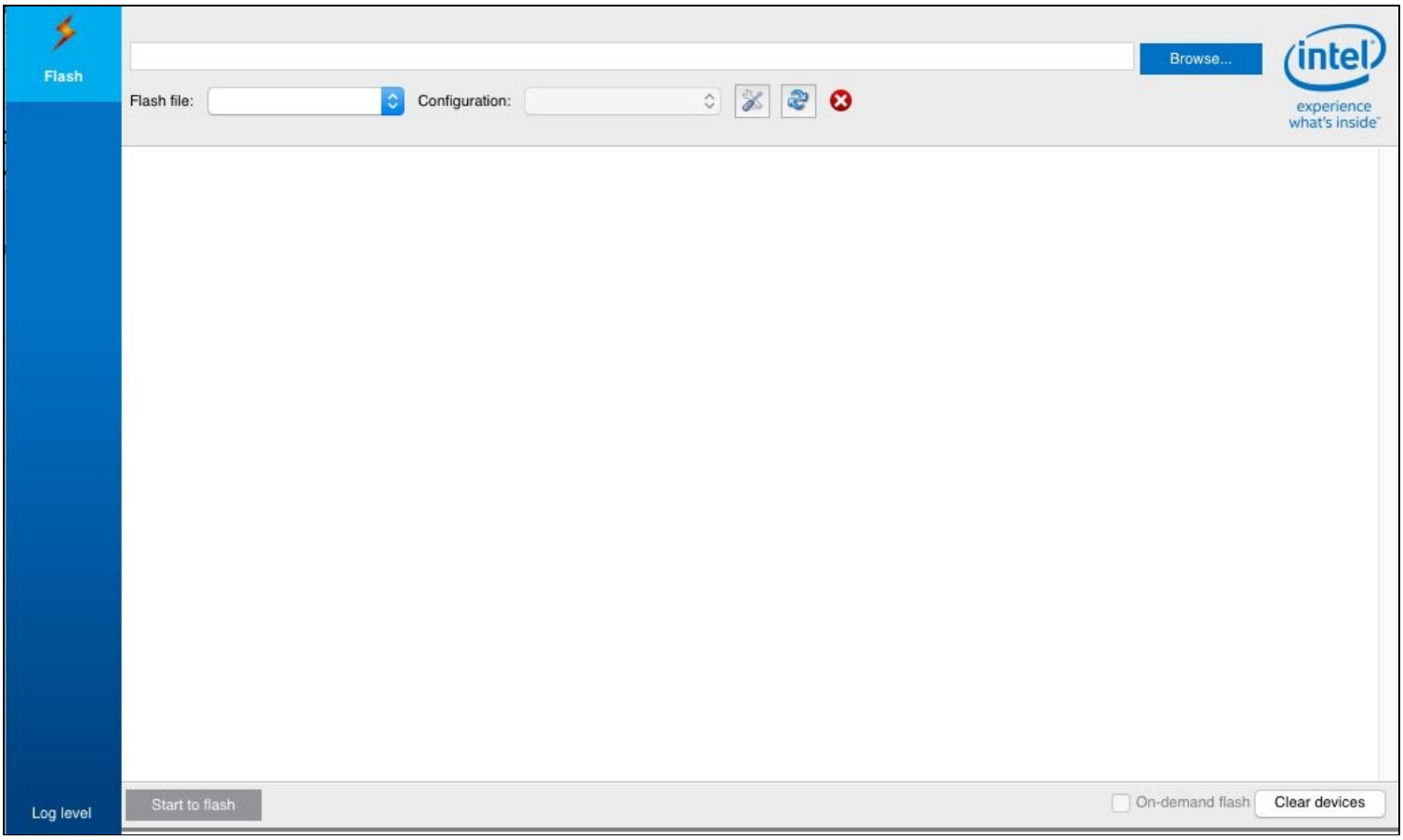
Moreover, check if the *fastboot* application is installed on your system. To do so, go to `<Android_SDK_HOME>/platform-tools`.

If you do not have the SDK installed correctly, please go to SDK Manager at https://developer.android.com/tools/help/sdk-manager.html and download and install it before continuing the flashing process.

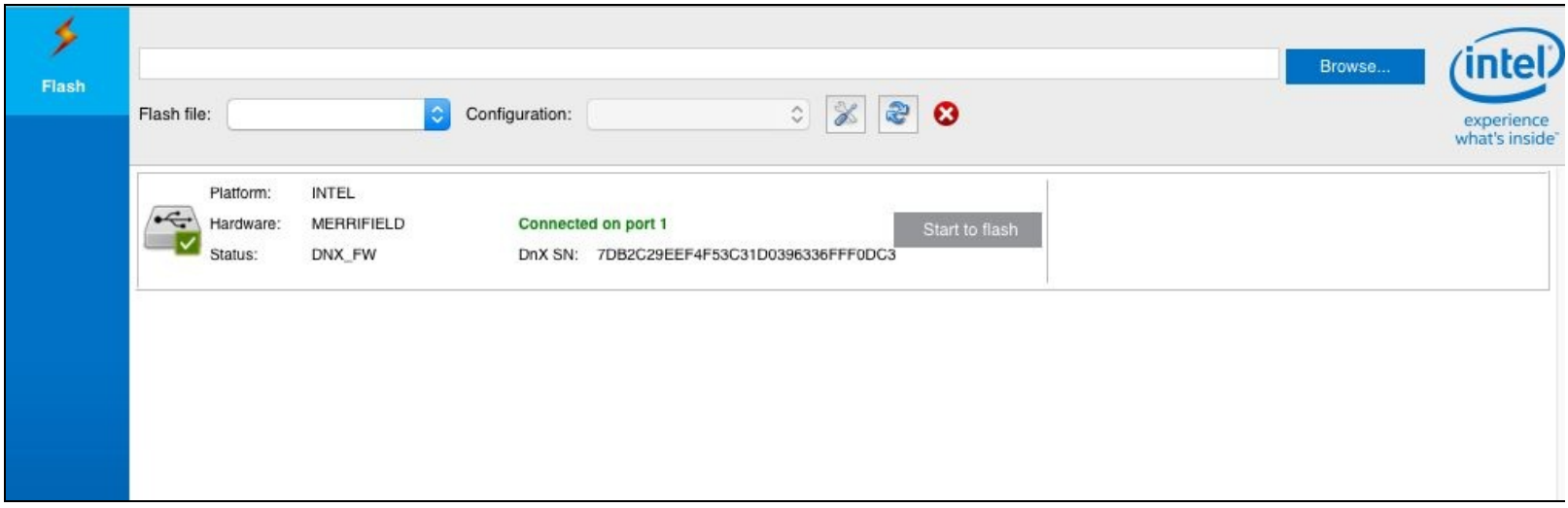
Let us start:

1. Go to https://developer.android.com/things/preview/download.html and download the image for Intel Edison.
2. Unzip the file.
3. Go to https://01.org/android-ia/downloads/intel-platform-flash-tool-lite. Download and install Platform flash tool light according to your operating system (OS X or Windows).

4. In the directory where you unzipped the image downloaded at step 1 there is a file called `FlashEdison.json`. This is our file. Check if it exists before continuing.
5. Run the Platform flash tool light:

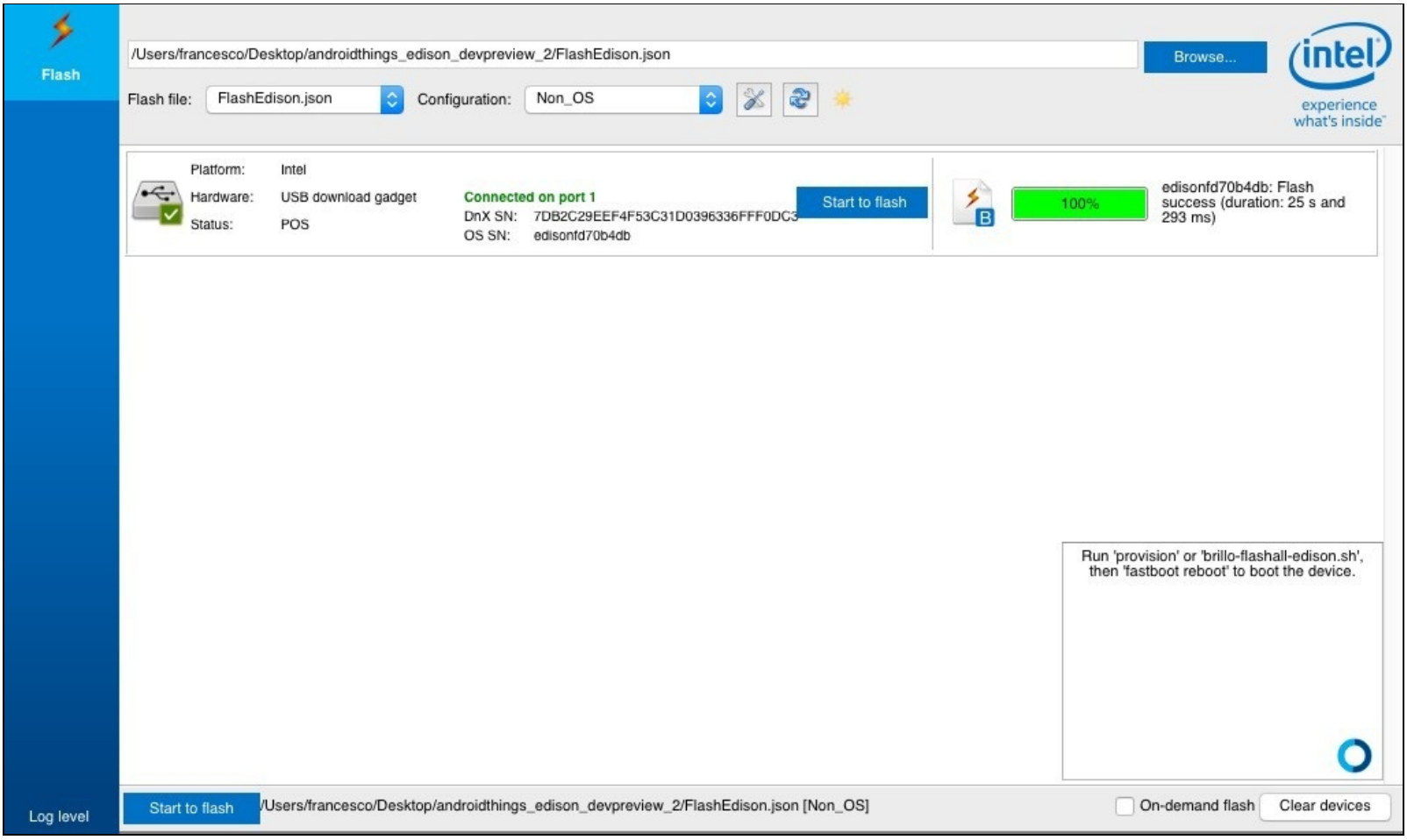


6. If you are using Intel Edison with Arduino breakout kit be sure that you:
    1. Click on the *FW* button and keep it pressed until step 7.
    2. Connect the USB port (J16) to your PC or Mac.
7. When the board is connected to your PC or Mac, it appears in the Platform Flash Tool Light:



8. Click on the Browse button and select the `FlashEdison.json` file, as described in step 4.
9. Check in Platform Tool Flash Light that the Configuration list box contains `Non_OS`.

10. Click on the Flash button and wait for the end of the process, as shown in the following screenshot:



11. Open a terminal console or the command prompt and execute the following command:

```
<Android_SDK>/platform-tools/adb reboot bootloader
```

12. To verify that the board is connected, write the following:

```
<Android_SDK>/platform-tools/fastboot devices
```

You should get the following as the result: `edisonXXXXX`

13. Move to the directory containing the unzipped content.

14. Write these commands:

```
<Android_SDK>/platform-tools/fastboot
flash gpt partition-table.img
flash u-boot u-boot-edison.bin flash boot_a boot.img
flash boot_b boot.img flash system_b system.img
flash userdata userdata.img erase misc
set_active _a
```

Now wait until the process is complete.

15. As the process completes and you have the prompt again, execute the following:

```
<Android_SDK>/platform-tools/fastboot
flash gapps_a gapps.img
flash gapps_b gapps.img
```

Wait until the end of the process.

16. Finally, execute the last command:

```
<Android_SDK>/platform-tools/fastboot
flash oem_a oem.img
flash oem_b oem.img
```

17. At the end, reboot your board:

```
<Android_SDK>/platform-tools/fastboot reboot
```

You can verify your installation listing the Android device connected to your system with:

```
adb devices
```

In the device list, there should be a device named *edison*.

adb shell am startservice
-n com.google.wifisetup/.WifiSetupService
-a WifiSetupService.Connect
-e ssid <Your_WIFI_SSID>
-e passphrase <WIFI_password>

Where `Your_WIFI_SSID` is the ID of your `WIFI` and `WIFI_password` is the password you use to connect to your WiFi.

Creating the first Android Things project

Considering that Android Things derives from Android, the development process and the app structure are the as same we use in a common Android app. For this reason, the development tool to use for Android Things is Android Studio. If you have already used Android Studio in the past, reading this section will help you to discover the main differences between an Android Things app and an Android app. Otherwise, if you are new to Android development, this section will guide you step by step to create your first Android Things app.

Android Studio is the official development environment to develop Android Things apps, therefore, before starting, it is necessary you have installed it. If not, go to https://developer.android.com/studio/index.html, to download and install it. The development environment must adhere to these prerequisites:

- SDK tools version 24 or higher
- Update the SDK with Android 7 (API level 24)
- Android Studio 2.2 or higher

If your environment does not meet the previous conditions, you have to update your Android Studio using the Update manager.

Now there are two alternatives to starting a new project:

- Clone a template project from GitHub and import it into Android Studio
- Create a new Android project in Android Studio

To better understand the main differences between Android and Android Things you should follow option number 2, at least the first time.

Cloning the template project

This is the fastest path because with a few steps you are ready to develop an Android Things app:

1. Go to https://github.com/androidthings/new-project-template and clone the repository. Open a terminal and write the following:

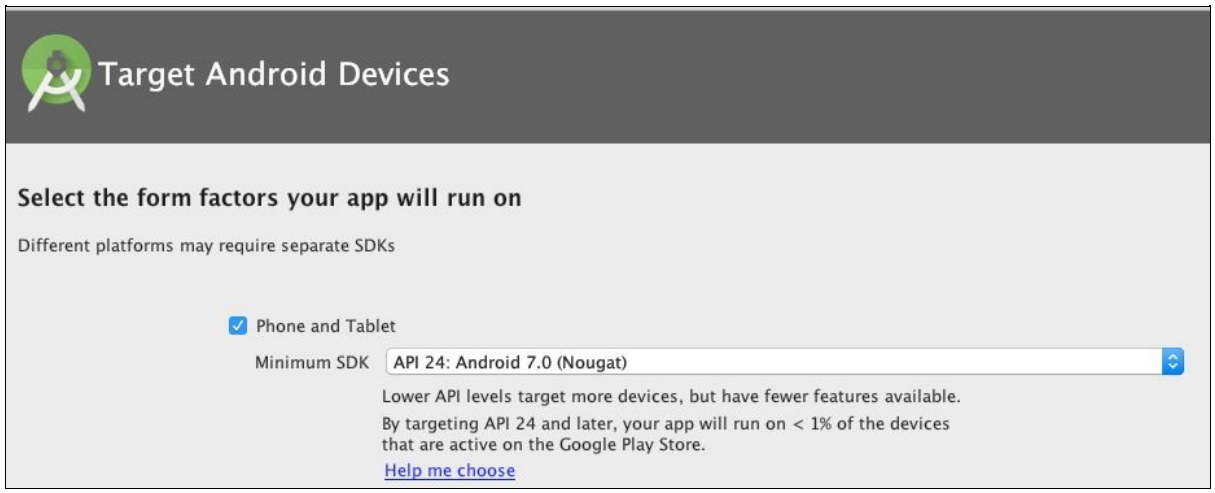
```
git clone https://github.com/androidthings/new-project-template.git
```

2. Now you have to import the cloned project into Android Studio.

Create the project manually

This step is longer in respect to the previous option, but it is useful to know the main differences between these two worlds:

1. Create a new Android project. Do not forget to set the Minimum SDK to level API 24:



2. By now, you should create a project with empty activity. Confirm and create the new project.

There are some steps you have to follow before your Android app project turns into an Android Things app project:

1. Open the `Gradle` scripts folder and modify `build.gradle` (app-level) and replace the dependency directive with the following lines:

```
dependencies {
  provided 'com.google.android.things:androidthings:     0.2-devpreview' }
```

2. Open the `res` folder and remove all the files under it except `strings.xml`.
3. Open `Manifest.xml` and remove the `android:theme` attribute in the application tag.
4. In `Manifest.xml` add the following line inside the application tag:

```
<uses-library android:name="com.google.android.things"/>
```

5. In the `layout` folder, open all the layout files created automatically and remove the references to values.
6. In the activity created by default (`MainActivity.java`) remove this line:

```
import android.support.v7.app.AppCompatActivity;
```

7. Replace `AppCompatActivity` with `Activity`.
8. Under the folder `java` remove all the folders except the one with your package name.

That's all. You have now transformed an Android app project into an Android Things app project. Compiling the code you will have no errors. In future, you can simply clone the repository holding the project template and start coding.

Differences between Android and Android Things

As you can see an Android Things project is very similar to an Android project. We always have Activities, layouts, gradle files, and so on. At the same time, there are some differences:

- Android Things does not use multiple layouts to support different screen sizes. So when we develop an Android Things app we create only one layout.
- Android Things does not support themes and styles.
- Android support libraries are not available in Android Things.

Create your first Android Things app

In this paragraph, we will modify the previous project and we will see how to control peripherals connected to Android Things. In more details, we will control the RGB led color using the three buttons. Each button controls a color (Red, Green, and Blue), so that when you press one button, Android Things turns on and off the corresponding color on the RGB led. To create this project, you need:

- Wire jumpings
- Resistors (200Ohm, 10Kohm)
- Three buttons

The following image shows the button that we will use in the project:

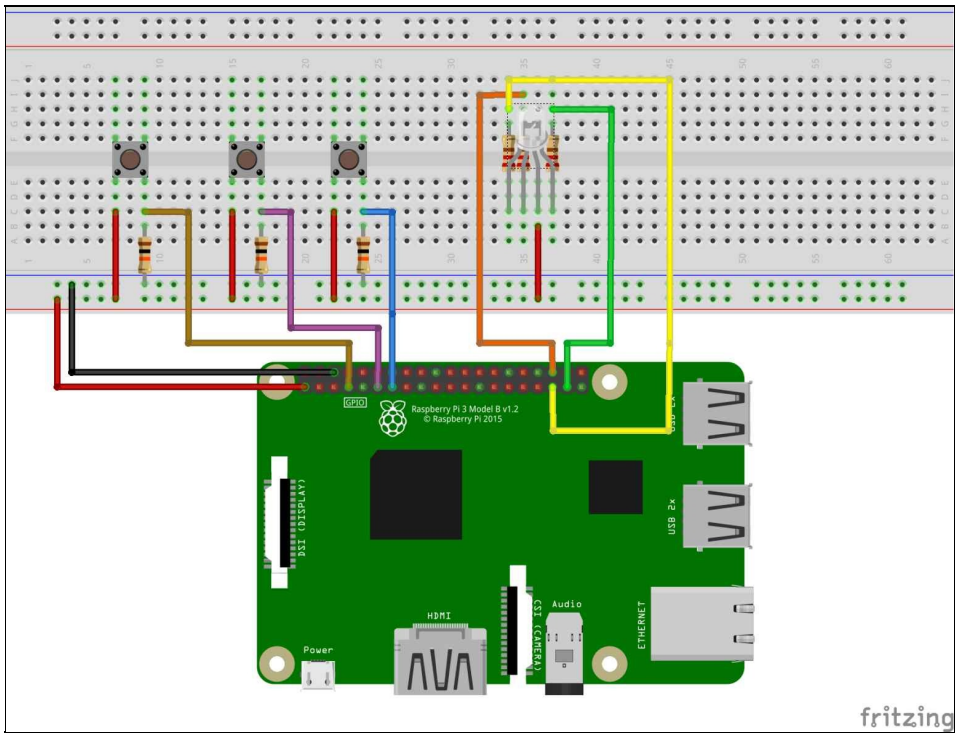
Source: https://www.sparkfun.com/products/97

The following image shows a 1 RGB Led:

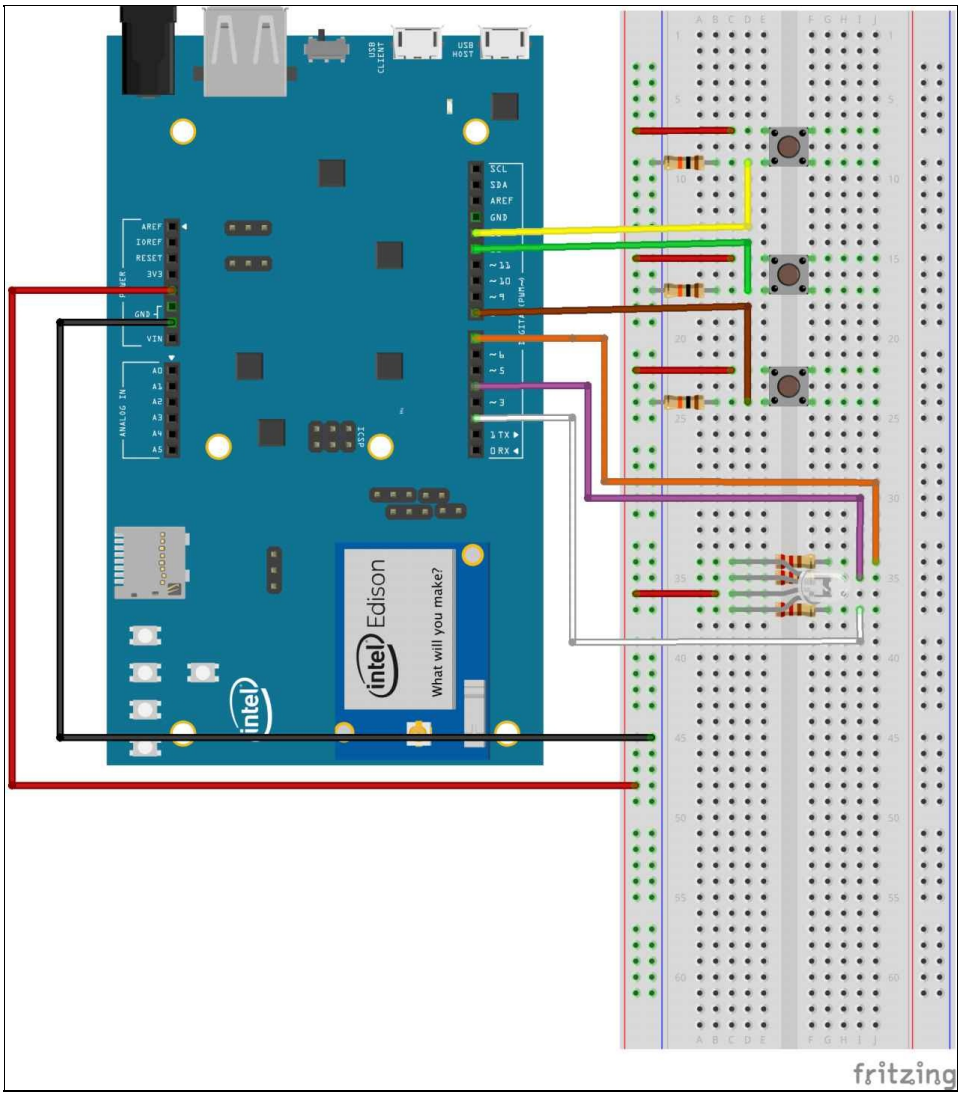
Source: https://www.sparkfun.com/products/10820

Before connecting the devices and resistor to the board be sure that the board is disconnected from the PC, otherwise you could damage it.

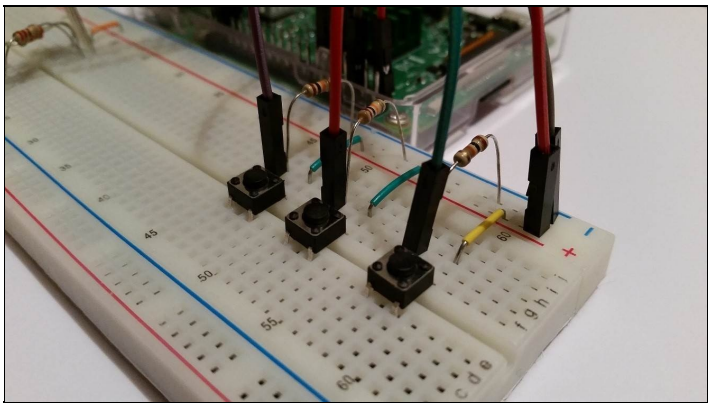
The following figure describes how to connect these components to the Raspberry Pi 3:



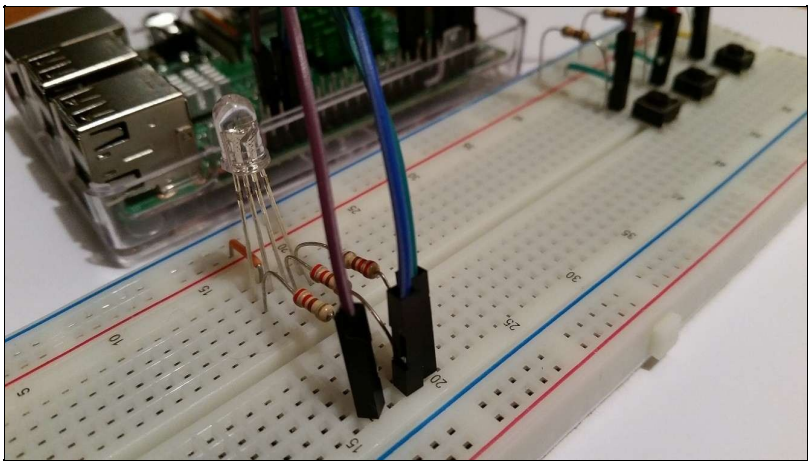
If you are using Intel Edison the schema is as follows:



The following image shows how to connect buttons in practice:



The connections are quite simple; a pull-down resistor of 10 Kohm connects one button pin to the ground. Each button has a pull-down resistor. Moreover, the following image shows how to connect the led:



A 200 Ohm resistor connects each RGB led pin and the boards pin limiting the current flowing into the LED. The other pin, the anode pin, is connected to 3.3V for Raspberry Pi and +5V for Intel Edison. Before modifying the source code, it is necessary to add a library that helps us to interact with buttons easily. Open `build.gradle` (app-level) and modify the file adding the following lines:

```
dependencies {
...
compile 'com.google.android.things.contrib:driver-button:0.1'
}
```

Using this library, we can handle the button status. Moreover, we can create listeners to listen to button state changes.

Now open the `MainActivity.java` that you created in the project and add the following lines:

1. In the `onCreate` method add the following:

```
        PeripheralManagerService manager = new PeripheralManagerService();
```

   This one of the most important classes introduced by Android Things SDK. By now, you should know that this class is used to interact with external peripherals. It exposes a set of methods to interact with several devices using different protocols (that is, GPIO, PWM, and so on). Using this class, an Android Things app turns on or off each board pin, so that it can control the external devices, or it can open a port for a specific purpose.

2. Create three different instances of the `Button` class corresponding to each button used in the circuit:

```
Button button1 = new Button("IO13", Button.LogicState.PRESSED_WHEN_LOW);
```

One important thing to notice is that we have to specify the pin where the button is connected to the board. In the following code line, the pin is IO13.

By now, it is enough to know that each pin on the board has a specific name and these names change depending on the board. For example, if you use Intel Edison, the pin names are different from Raspberry Pi 3 pin layout. We will cover this aspect in the next chapter. The other parameter represents the button logic level when it is pressed. If you are using Raspberry Pi 3, then instead of the code line shown previously, you have to use the following:

```
Button button1 = new Button("BCM4", Button.LogicState.PRESSED_WHEN_LOW);
```

Maybe you are wondering if there are some compatibility problems when we install an Android Things app on different boards. The answer is yes, but we will see in the next chapter how to handle this problem and how to create an app that is board-independent:

1. Now add a listener to be notified when the user presses the button. We do it as if this is an Android app with a UI:

```
button1.setOnButtonEventListener(
  new Button.OnButtonEventListener() {
    @Override
    public void onButtonEvent(Button button, boolean
    pressed) { if (pressed) {
      redPressed = !redPressed;
      try {
        redIO.setValue(redPressed);
        }
        catch (IOException e1) {}
    }
  }
});
```

The interesting part is that we set the `redIO` pin value to 1 (high) or 0 (low) according to the button status. The `redIO` represents the pin that connects the red pin of the led. We get the reference to it using the following:

```
   redIO = manager.openGpio("IO2");
```

Do not worry now about this piece of code; we will cover it in the next chapter. Using the preceding code line, we open the communication to the LED using another board pin. The previous example is for Intel Edison, and again if you are using Raspberry, the pin name changes.

2. Now repeat the same piece of code shown previously for the green and blue buttons:

```
button2.setOnButtonEventListener(new Button.OnButtonEventListener()
{
    @Override
    public void onButtonEvent(Button button,
    boolean pressed) {
        if (pressed) {
         greenPressed = !greenPressed; try {
```

```
                    greenIO.setValue(greenPressed);
                   }
                catch (IOException e1) {}
              }
            }
        });
```

3. Where `greenIO` is defined as follows:

```
        greenIO = manager.openGpio("IO4");
```

4. While for the blue buttons:

```
        button3.setOnButtonEventListener(new Button.OnButtonEventListener()
        {
            @Override
            public void onButtonEvent(Button button,
            boolean pressed) {
                if (pressed) {
                    bluePressed = !bluePressed; try {
                        blueIO.setValue(bluePressed);
                        }
                    catch (IOException e1) {}
                }
            }
        });
```

5. And the `blueIO` is defined as follows:

```
        blueIO = manager.openGpio("IO7");
```

6. Finally, we have to modify `Manifest.xml`. From the Android point of view, an app uses the `Manifest.xml` to define the Android components such as Activity, Services, and so on.

This is still valid in the Android Things project, but there is a difference in the way it declares an Activity:

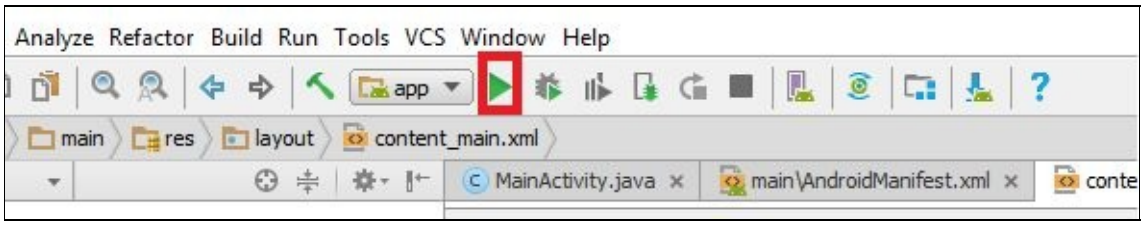
1. Open the `Manifest.xml` and look for the Activity definition.
2. Remove all the intent-filter tag.
3. Add the following lines at the same position:

```
        <intent-filter>
        <action
        android:name="android.intent.action.MAIN" />
        <category android:name=
        "android.intent.category.IOT_LAUNCHER" />
        <category android:name=
        "android.intent.category.DEFAULT" />
        </intent-filter>
```

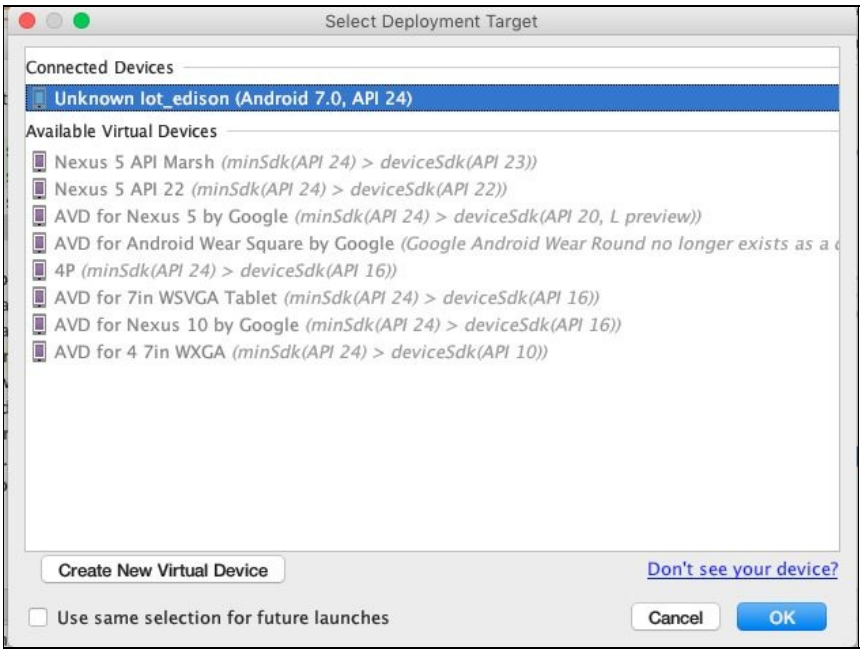
4. Save the file.

The interesting part to notice is a new category type. If we want to execute an Activity that runs on an embedded device such as Raspberry or Intel Edison we have to add the category `IOT_LAUNCHER`.

That's all. Now you can connect the board to your PC/Mac. Press the run button at the top of Android Studio:



And wait until the board appears in the list of available devices, as shown in the following screenshot:



Now you can execute the app. The installation process is the same as used for the Android app. When the process completes you can start using the app.

When you press each button, you should see the led changing color, moreover, you can completely turn off the led.

Summary

This chapter introduced you to Android Things and how it works. We installed Android Things on Raspberry Pi 3 and Intel Edison. This was a necessary step so that we have a development board where we can test our next Android Things IoT projects. We developed and tested our first Android Things app that interacts with external peripherals. Now, you are ready to start developing amazing IoT projects using Android Things SDK.

In the next chapter, we will build an alarm system. We will use a PIR sensor with Android Things to detect motion. Moreover, you will explore how to use GPIO pins to communicate with the external world.

Creating an Alarm System Using Android Things

In this chapter, we will build an alarm system using Android Things. The target of this project is creating a system that detects movements using PIR sensors and when this event happens the Android Things app will send a notification to the user smartphone. The principles of this project are commonly used in real alarm systems we have in our homes, but we will build it with a totally new operating system. This is an interesting project because it uses, at the same time, sensors and cloud platforms. Through this project, we will explore how to use GPIO pins in Android Things and how to interact with two states sensors.

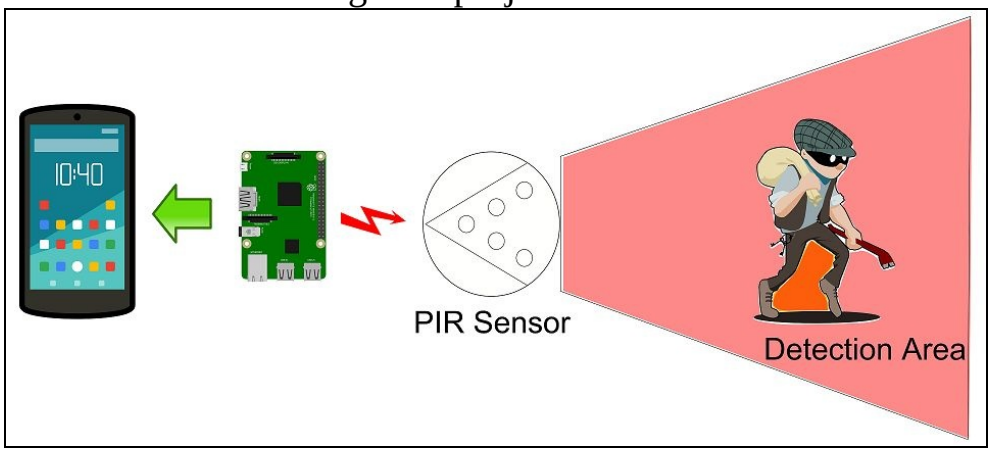
The main topics covered in this chapter are:

- How to use build an alarm system
- How to use GPIO pins and PIR sensors
- How to handle events from a GPIO pin
- How to build an app that is independent of the board
- How to notify events from Android Things to Android smartphones

This project demonstrates how powerful Android Things SDK is and how we can build an IoT project using our Android expertise. Let us start describing the project that we will build.

Alarm system project description

An alarm system is a complex system that uses several sensors to keep our home safe. At the heart of these types of systems, there are sensors that are able to detect motion. In other words, these sensors can detect if an object is moving in their detection area. When this happens, they notify this event. In this chapter, we will create a real-life project that uses these sensors to detect motion and notifies the event to the smartphones. At the end of this project, we will be able to detect if someone is entering our home without our authorization. Once you have built this project, you can expand it, adding more sensors so that you can monitor several rooms. Moreover, this project can be used as a starting point and can be expanded, adding new features as we will see in the following sections. The following figure describes how this Android Things IoT project will work:



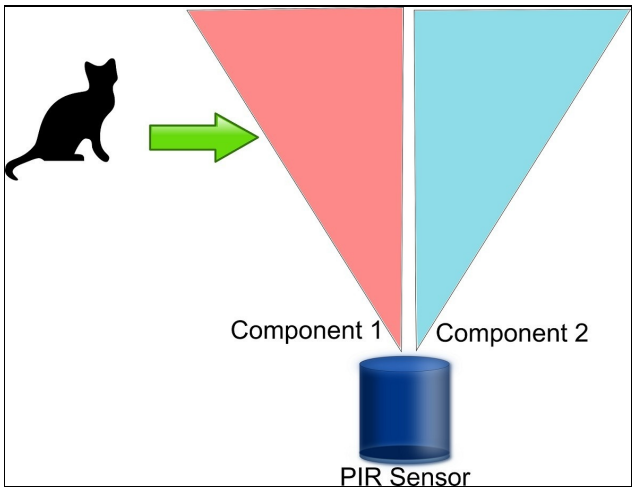
The following are the main steps:

1. The PIR sensor scans the detection area looking for movement.
2. As soon as it detects a movement it notifies the event to our Android Things board.
3. The Android Things board handles the notification event and contacts *Google Firebase* to send a message to the user's smartphone.

PIR sensor

In the previous paragraph, we talked about PIR sensors and it is useful to describe these briefly so that we have a common base to start our project. PIR sensor stands for a **Passive InfraRed** (**PIR**) sensor. This is a class of sensors that are able to detect movement by measuring the **infrared** (**IR**) light emitted by an object. All objects especially human bodies, animals, and so on emit energy using infrared rays. This type of energy is not visible to human eyes, but we can measure it using special sensors like this one. As a matter of fact, what we really measure is a variation in the emitted energy. The passive term refers to the fact that this sensor does not produce or radiate infrared rays, but it simply detects the energy emitted. Before digging into the project it is convenient to know how it works to better understand how to use it in the right way. A PIR sensor is a quite complex sensor that uses two different components. Each component is sensitive to infrared rays, as described previously.

The following figure describes how a PIR sensor works:



When a warm body (like the cat in the picture) passes through the detection area, the first component gets excited while the second component remains idle. As the body moves and leaves the first component detection area, the first component gets idle and the second component gets excited. Using this simple principle the sensor can detect when a body is moving. At the end of this process, an event is triggered. PIRs have several configurations with different features.

The most common model is the one that uses F*resnel lenses* that help to widen the detection area. The following image shows the PIR sensor that we will use in this Android Things project:

The sensor has two potentiometers:

- One to adjust the sensitivity
- The other one to control the time the signal is high when an object is detected

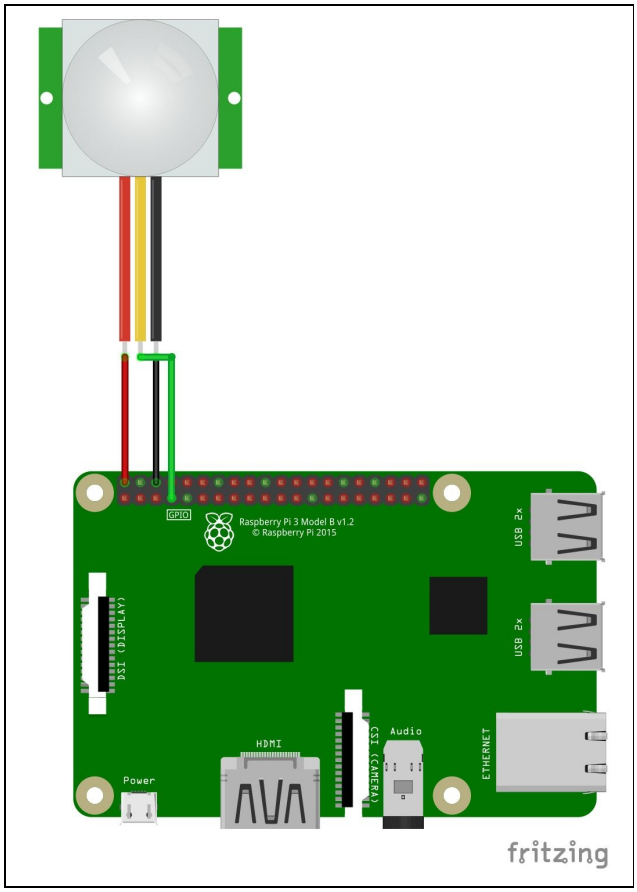
Project schematic

The peripherals we need to build this project are:

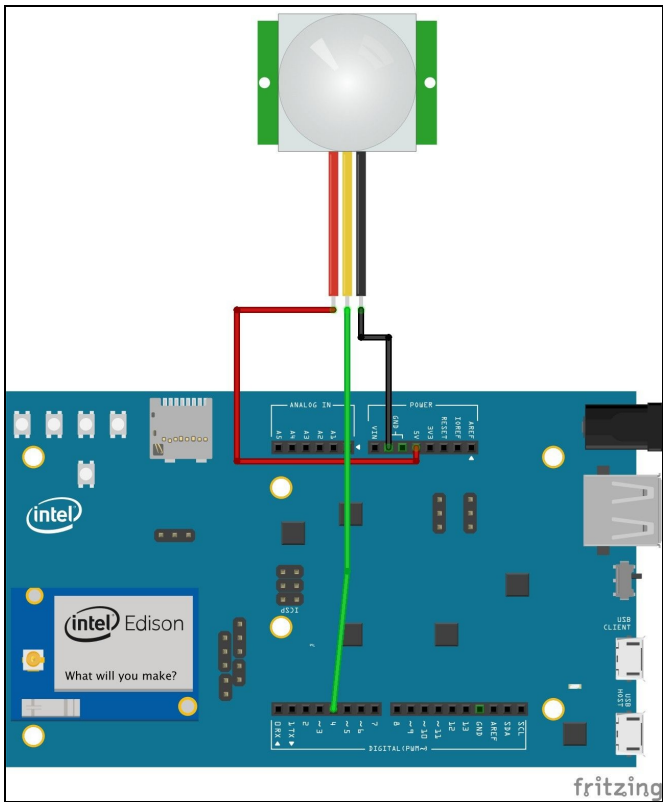
- PIR sensor
- Raspberry Pi 3 or Intel Edison with Arduino breakout kit
- Google Firebase account
- Wire jumpings

You can buy a PIR sensor in an online store such as Amazon, Sparkfun, or Adafruit.

The following diagram describes how to connect the PIR sensor to the Android Things board if you are using Raspberry Pi 3:



If you are using Intel Edison with Arduino breakout kit the connections are shown here:



In this case, we can directly connect the sensor to our board. The connections are:

- Connect the PIR to the pin that supplies +5V
- Connect the ground PIR pin to the ground of the board
- Connect the PIR signal to pin 7 for Raspberry or pin 4 for Intel Edison

As you can notice, the sensor connects to +5V in both cases: Raspberry and Intel Edison. The signal pin provided by PIR is zero when no motion is detected and +3V when the motion is detected. Considering that the high level supplied by the PIR is +3, then we can connect the PIR sensor safely to Raspberry Pi 3.

Remember to connect the sensor to the board when this one is unplugged from your computer. Do not try to connect the sensor when your board is turned on, as you could damage your board and the sensor.

How to use GPIO pins

When we connect peripherals to an Android Things board we use pins. There several types of pins. This project uses GPIO pins. **GPIO** stands for **General Purpose Input Output**. These pins are the interface between the board (such as Raspberry or Intel Edison) and the world. You can think of them as a switch that can be turned on or off. Using GPIO pins we handle binary devices. A GPIO pin can have only two states:

- On or High level
- Off or Low or zero

According to the nature of these pins, we can connect to these pins all the peripherals that have two states. Typical examples of these peripherals are switches or simple LEDs (only one color led). The PIR sensor described previously belongs to this category.

Android Things SDK provides an important class that helps us to interact with GPIO pins hiding the communication details. This class is called `PeripheralManagerService`. Using `PeripheralManagerService` we can do several actions on the pins:

- Get the pins list
- Get the pin state
- Set the pin state

These are the main actions; anyway, this class provides several methods that help us to manage the pin connection and its state.

To use a GPIO pin in Android Things we have to follow three steps:

1. Get an instance of `PeripheralManagerService`.
2. Open the connection to the pin using the pin identifier.
3. Declare if the pin is used to read (input) or to write (output).

Let us see how we can implement it in our project.

Clone the Android Things project template as described in the previous chapter. Open `MainActivity.java` and in the `onCreate` method add the following:

```
PeripheralManagerService service = new PeripheralManagerService();
```

In this way, we get an instance of `PeripheralManagerService`, the class that handles the GPIO communication details.

In the `onCreate` method add the following lines:

```
try {
   gpioPin = service.openGpio(GPIO_PIN);
   gpioPin.setDirection(Gpio.DIRECTION_IN);
   gpioPin.setActiveType(Gpio.ACTIVE_HIGH);
```

```
}
catch(IOexception ioe) {}
```

The preceding code is simple: the app opens a connection to a GPIO pin specified in `GPIO_PIN`. By now you should remember that Raspberry Pi 3 and Intel Edison have a different GPIO pinout. According to the schema shown previously, the pin names are:

- BMC 4 for Raspberry
- IO4 for Intel Edison with Arduino breakout kit

After that, the app specifies the type of connection it will handle. In this project, we want to read from the pin so we declare `Gpio.DIRECTION_IN`. As we said before, a GPIO pin can be used to read or write so there are two possible values:

- `Gpio.DIRECTION_IN` if we read
- `Gpio.DIRECTION_OUT` if we write

Finally, we have to set if we want that `true` value corresponding to the high voltage level or zero level. We do it using `setActiveType` that accepts two values:

- `Gpio.ACTIVE_HIGH`: The value is *true* if the pin is at high voltage
- `Gpio.ACTIVE_LOW`: The value is *true* if the pin is at low voltage or zero

All these methods can raise an `IOException` when a problem occurs. For this reason, they are in a try/catch clause.

Using GPIO pins we can build amazing IoT projects with Android Things. Anyway, when you connect peripherals to your board using GPIO pins be sure that the output of the peripheral is compatible with the board operating voltage. If the output sensor voltage is higher than the board operating voltage, you could damage the board.

Reading from the GPIO pin

Once we have initialized the GPIO pin connection we can start reading its state. To read the state of the sensor we use:

```
boolean status = gpioPin.getValue();
```

The `getValue` method returns `true` or `false`. Using this method we can know the pin state. In this project, we want to check the state of the pin constantly to know if someone is moving in our room. The simplest way to do it is by creating a thread and continuing to read the pin state.

To do it, open the `MainActivity.java` and add the following lines at the end of the `onCreate` method:

```
(new Thread(new Runnable() {
    @Override
    public void run() { try {
        while (true) {
          boolean status = gpioPin.getValue();
          Log.d(TAG, "State [" + status + "]");
          if (status) {
            Log.i(TAG, "Motion detected...");
            }
            Thread.sleep(5000);
          }
        }
        catch(Exception e) { e.printStackTrace();
      }
    }
})).start();
```

Now try to run the app as you are used to doing in Android. Open the log window, and notice that the app keeps on writing the state of the sensor. If the state is `false` then no motion is detected.

Try to move your hand in front of the sensor and you will see how the PIR sensor will detect you. The following lines shows the app log when moving our hand. As you can notice from the log shown here, the PIR detects the movement and writes a message:

```
"Motion detected":
androidthings.project.alarm D/MainActivity: Sensor status [false] androidthings.project.alarm D/MainActivity: Sensor
```

Even if this is a working approach, it's time consuming because our Android Things app has to monitor the pin state all the time, even if the PIR sensor is not detecting anybody. Fortunately, there is another approach that uses `listeners`. This is less time-consuming and it is similar to the way we are used to developing an Android app.

How to add a listener to GPIO

As said before instead of reading the sensor state all the time we can use listeners. Android Things SDK provides a `callback` class that is invoked when the sensor changes its state. We can add a listener to a GPIO in two steps:

1. Declare the event we want to listen to.
2. Implement a callback class to handle the event and register it.

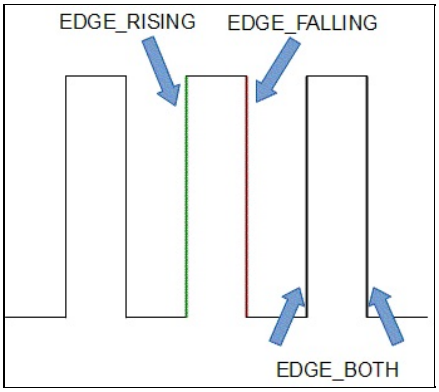
Let us describe each step.

Declare the event to listen to

The first step is defining the type of event we want to listen to. There are four different types of changing events:

- **EDGE_NONE**: No event is triggered.
- **EDGE_RISING**: Rising trigger. The pin voltage value is changing from low or zero to high or true.
- **EDGE_FALLING**: Falling trigger. The pin voltage value is changing from high or true to low or false.
- **EDGE_BOTH**: The combination of the last two changing events. In other words, we want to be notified when the signal changes from low to high or from high to low.

The following figure represents these types of events:



To declare the event we want to listen to, we use: gpioPin.setEdgeTriggerType(event_type);

Where `gpioPin` is an instance of the `Gpio` class. In this Android Things project, we want to be notified when the signal changes from low to high because it means we are detecting movement: gpioPin.setEdgeTriggerType(Gpio.EDGE_RISING);

Implementing the callback class

Once we have defined the event, we have to create a `callback` class that will handle the event when it will be raised. A `callback` class must extend `GpioCallback`. So in our project, the `callback` class will be:

```
private class SensorCallBack extends GpioCallback {
    @Override
    public boolean onGpioEdge(Gpio gpio) { try {
        boolean callBackState = gpio.getValue();
        Log.d(TAG, "Callback state ["+callBackState+"]");
        }
        catch(IOException ioe) { ioe.printStackTrace();
        } return true;
    }
    @Override
    public void onGpioError(Gpio gpio, int error) {
    super.onGpioError(gpio, error);
    }
}
```

At the end of `MainActivity.java`, just before the last right brace, add the class shown previously.

There are two important methods that we have to override in order to customize the behavior of the `callback` class:

- `public boolean onGpioEdge`
- `public boolean onGpioError`

The first one is invoked when the event we registered using `setEdgeTriggerType` is triggered. In our alarm system, we override this method to implement our custom logic. In this use case, this method is invoked only when the voltage of the pin rises from zero to high. As we will see in the next paragraphs, in our `callback` class, we will send a notification to the user smartphone.

The second method is invoked when an error occurs on the pin. We can use this class to gracefully handle the error and notify it to the user.

Finally, we have to register our `callback` class:

```
SensorCallBack callback = new SensorCallBack(); gpioPin.registerGpioCallback(callback);
```

That's all. We can run the app again and you can check that as soon as your hand moves in front of the PIR sensor the event is triggered. Opening the log, you will notice that the Android Things app logs *Call back state...* showing the sensor state.

How to close the connection with a GPIO pin

In this last step, we will learn how to close the connection with a GPIO pin. This is an important step because in this way, we free the resources and remove all the listeners we added to the GPIO pins.

An Android Things app has a life cycle very similar to an Android app. The place where we implement these actions is the Activity `onDestroy` method. In this method we have to:

- Remove all the listeners attached to the GPIO pins
- Close the connection to the GPIO pins

So, open `MainActivity.java` again and look for the `onDestroy` method and modify it: @Override
protected void onDestroy()
{ super.onDestroy(); Log.d(TAG, "onDestroy");
if (gpioPin != null) {
gpioPin.unregisterGpioCallback(sensorCallback);
try {
gpioPin.close(); gpioPin = null;
}
catch(Exception e) {}
}
}

Handle different boards in Android Things

There are two important aspects that we have not covered:

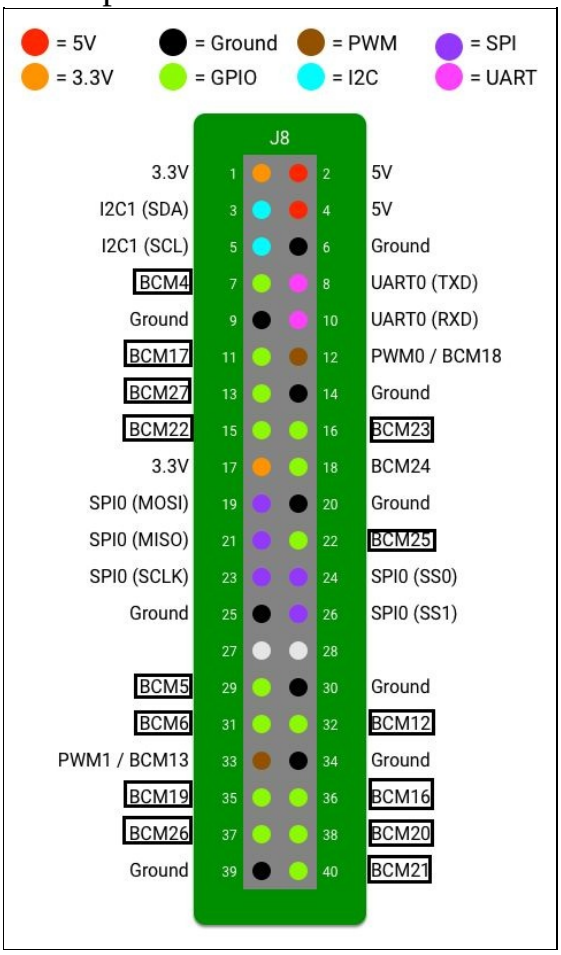
- How to select the pin to connect the peripherals to
- How to identify the pin name

Regarding the first aspect, in Raspberry Pi 3 and Intel Edison, but in general for all the boards, the pins do not provide the same features. In other words, we cannot connect the peripherals to a pin by choosing it randomly. We have to select the pin according to the peripheral specifications. In this context it is important to know the pinout of the boards so that we can identify the right pins for our peripherals.

The second aspect is relevant when we want to develop an Android Things app that will run on different boards. As we said in the previous chapter, from the code point of view, this is not a problem because the Java language at the base of Android Things SDK guarantees us that we can run it on all compatible boards. Until now, when we had to identify a pin, we used a **double** version, one for Raspberry Pi 3 and another one for Intel Edison. This works if we develop an app that runs on only one board, but if we want to build an app that is portable to different boards without changing the code, this approach would not work. In other words, we have to find a way to discover the board where the app is running and change the pin name according to it.

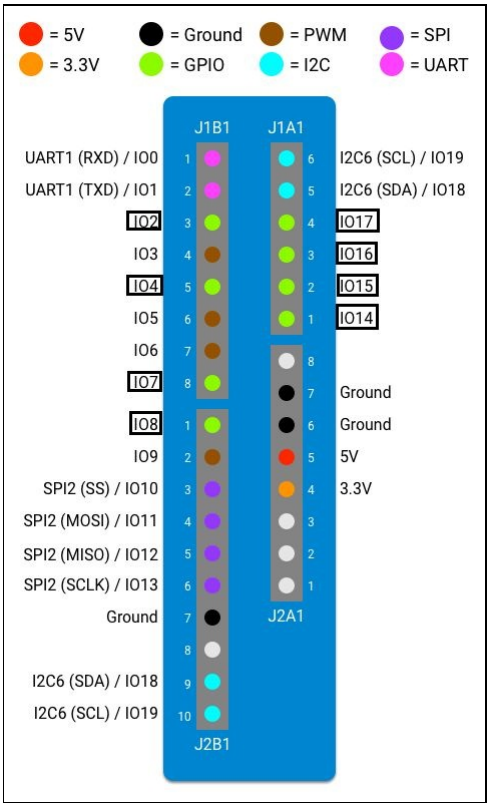
Android Things board pinout

For Raspberry Pi 3 the pinout is shown in the following figure. Notice that by now we are interested in GPIO pins only:



Source: https://developer.android.com/things/hardware/raspberrypi-io.html

For Intel Edison with Arduino breakout kit the pinout is:



Now it should be clear how we selected the pins in the project.

How to identify the board

In order to select the right pin names according to the board, we have to identify the board. Android Things SDK provides this constant:

```
Build.BOARD
```

Using this information, we can select the pin name at runtime in this way:

```
public class BoardPins {
    private static final String EDISON_ARDUINO =
    "edison_arduino"; private static final String
    RASPBERRY = "rpi3";
    public static String getPirPin() { switch
         (getBoardName()) {
            case RASPBERRY:
            return "BCM4";
            case EDISON_ARDUINO:
            return "IO4"; default:
            throw new IllegalArgumentException
            ("Unsupported device");
            }
        }
    private static String getBoardName() { String name =
     Build.BOARD;
    if (name.equals("edison")) { PeripheralManagerService
     service = new PeripheralManagerService();
    List<String> pinList = service.getGpioList(); if
     (pinList.size() > 0) {
    String pinName = pinList.get(0); if
     (pinName.startsWith("IO"))
    return EDISON_ARDUINO;
    }
}
return name;
}
}
```

The board name returned by Android Things SDK does not help us to distinguish between Intel Edison board variants. To this purpose, we list the pins and look for a specific name in the pin so that we are able to identify the board and its variant.

Notice that the name of the pin returned by this method is the name we get from the pinout shown in the preceding figures.

Now open MainActivity.java again and modify the method where we defined the pin. Look for:

```
gpioPin = service.openGpio...
```

And replace it with:

```
gpioPin = service.openGpio(BoardPins.getPirPin());
```

Now our Android Things app is independent of the board used to run the app.

All the time you reference a pin using its name, you have to use the approach shown previously to get its name so that your app will work on all the supported Android Things boards.

How to implement the notification system

Now we are ready to implement the last part of this project: the notification system. In the next paragraphs, we will describe how to send a notification to the user smartphone when motion is detected. As the messaging system, this IoT project uses *Google Firebase*. This is a cloud platform developed by Google providing several interesting services. We will use the Notification service.

There are several ways we can send a notification from the Android Things app to a user smartphone. To keep things simple, we will use *topic*. You can imagine a topic like a channel. After a device subscribes to a topic, it will receive all the messages published to this channel. In our project, the user smartphone behaves like a *subscriber* receiving messages from the channel, while the Android Things app behaves like a *publisher* publishing the messages.

Now it is clear the roles these two apps play in this project.

Before implementing it, we have to configure the Firebase Notification system.

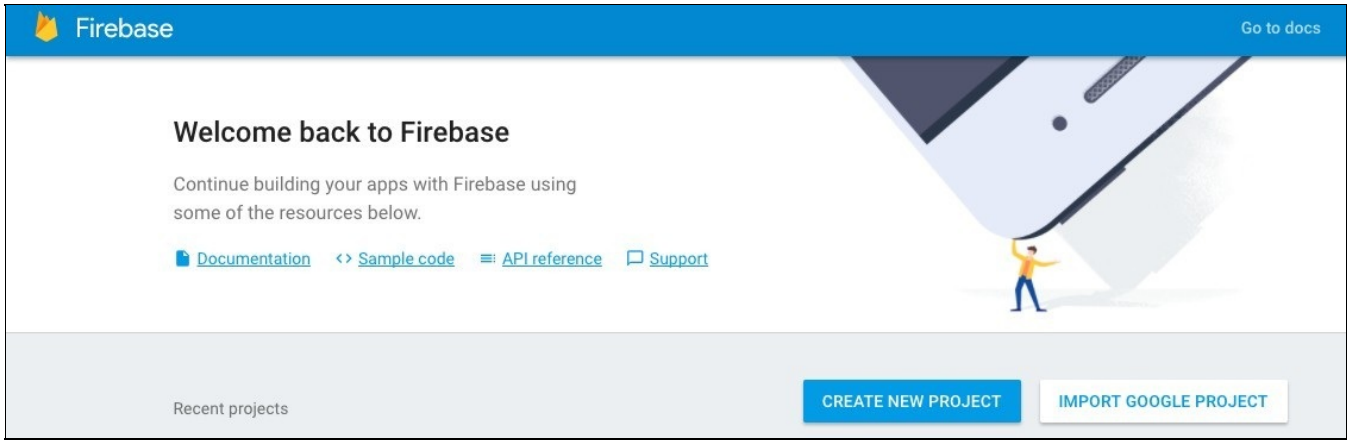
Configuring firebase

The first step is creating an account in Firebase:

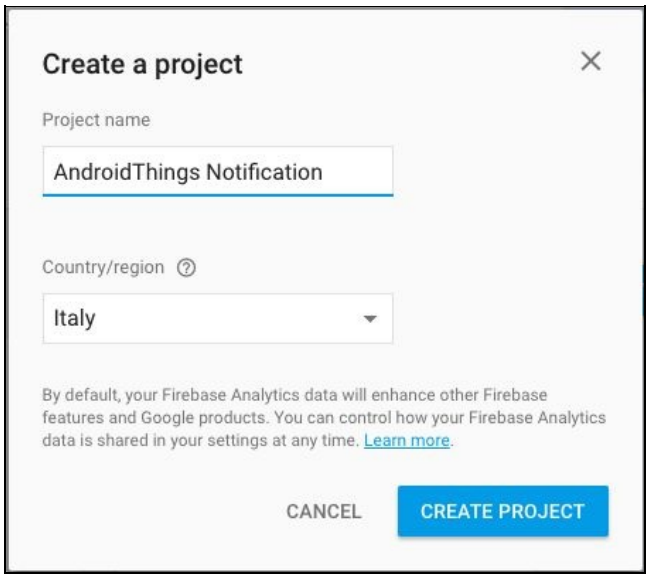
1. Go to the Firebase home page (https://firebase.google.com/) and click on Get started for free to create your account.
2. Provide all the information required.
3. Confirm and create the account.

Once your account is created, we can configure a new project:

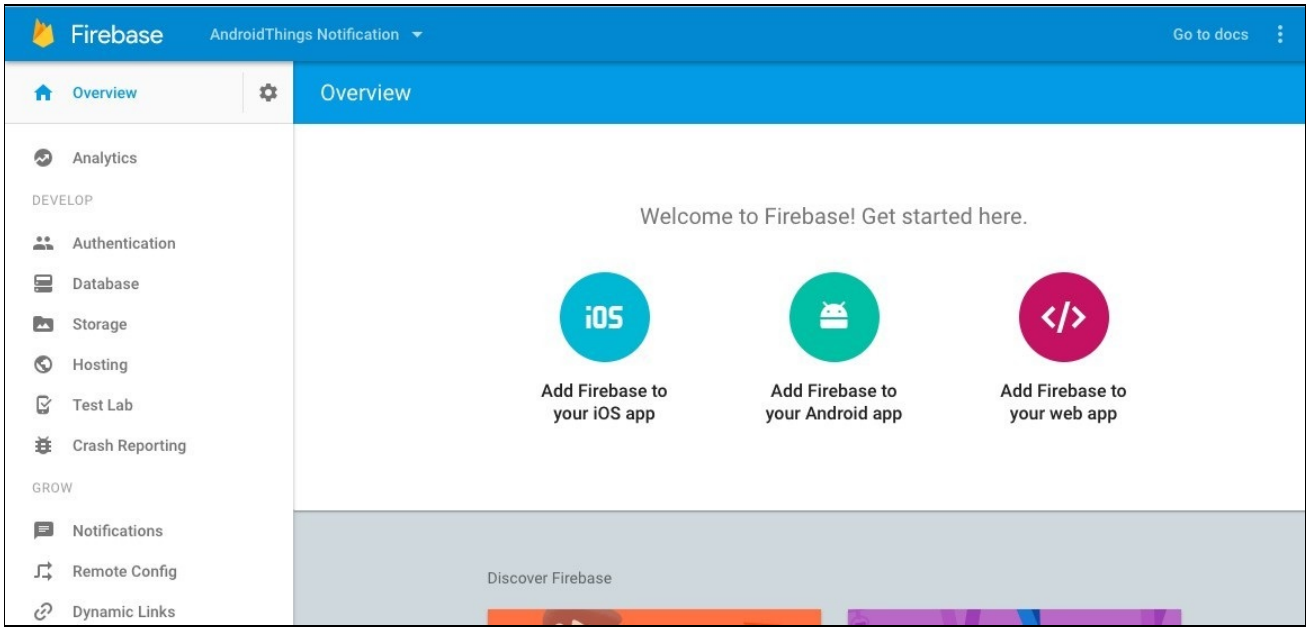
1. Log into the Firebase console.
2. Click on the link Go to console.
3. Now you can create the project:



4. Click on CREATE NEW PROJECT and you will get a page like this:

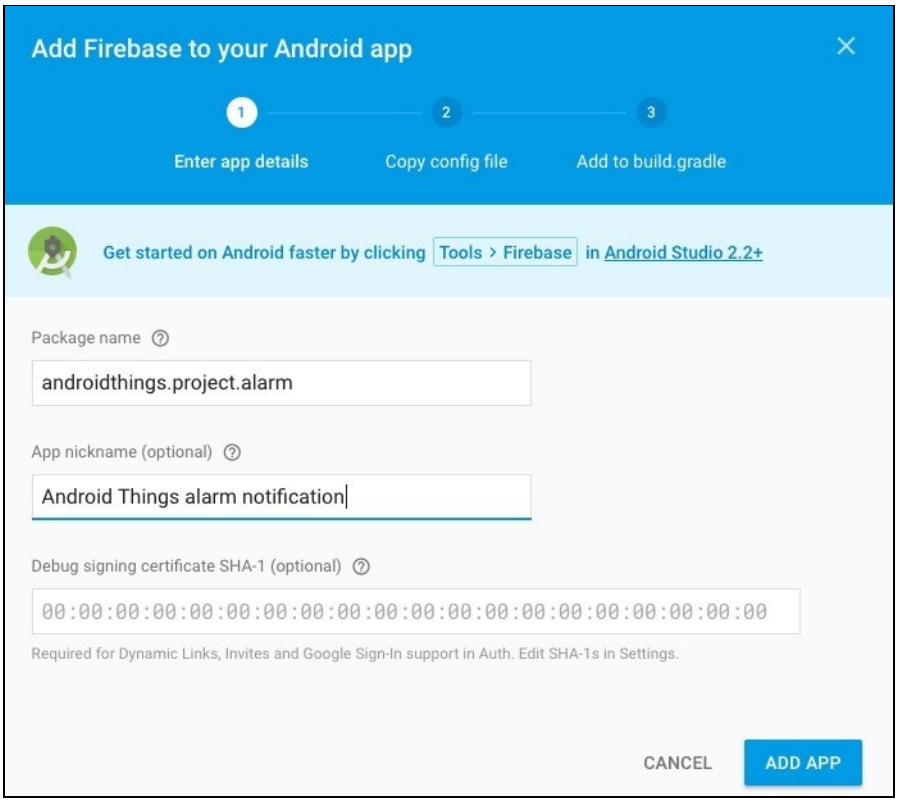


5. Provide the project name and the country/region and at the end create the project.
6. Once you have created your project you can manage it using the administration console:

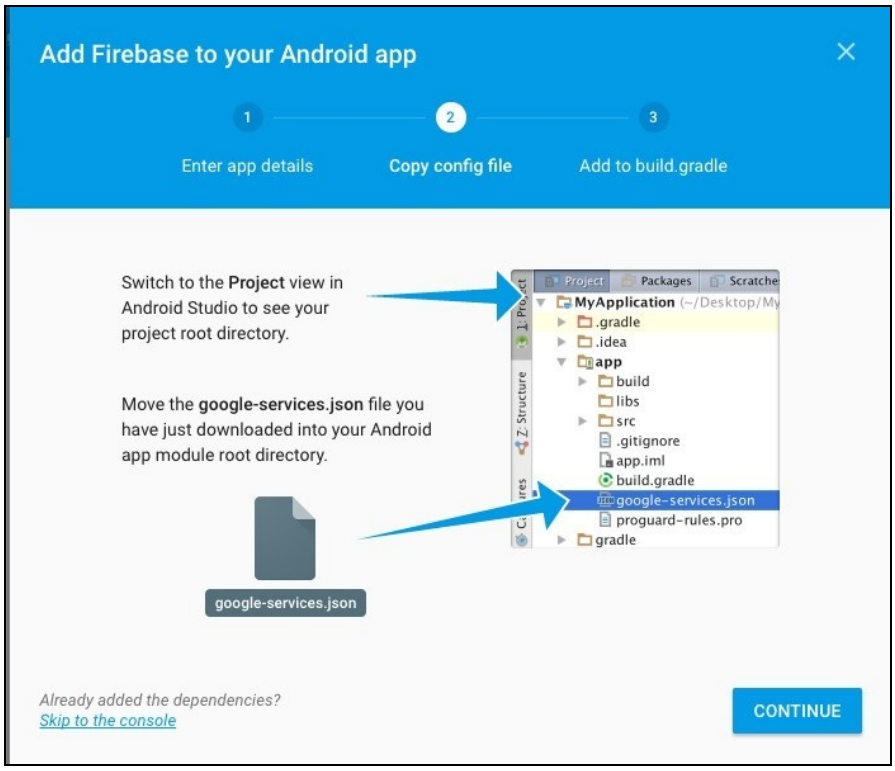


7. Now, we add our Android app to this project by clicking on Add firebase to your Android app. The Firebase console does not distinguish between Android and the Android Things app.

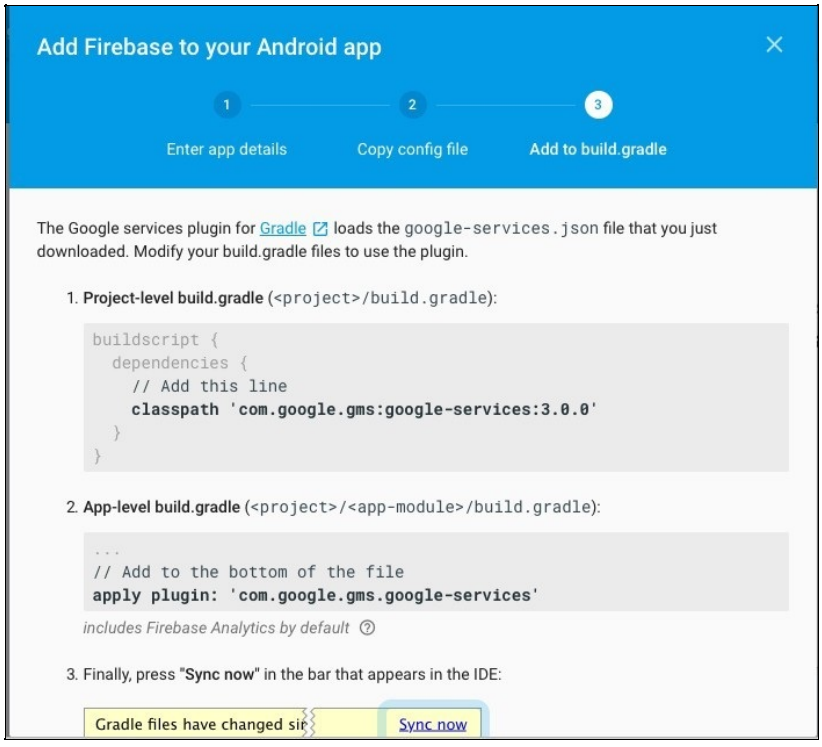
8. In the next screen, you have to add the Android Things app details. It is important that you provide the package name used in your project, as shown in the following screenshot:



9. Now click on Add app and you will be guided in the next two steps to configure your Android Things app, as shown in the following screenshot:



10. Finally, the last step:



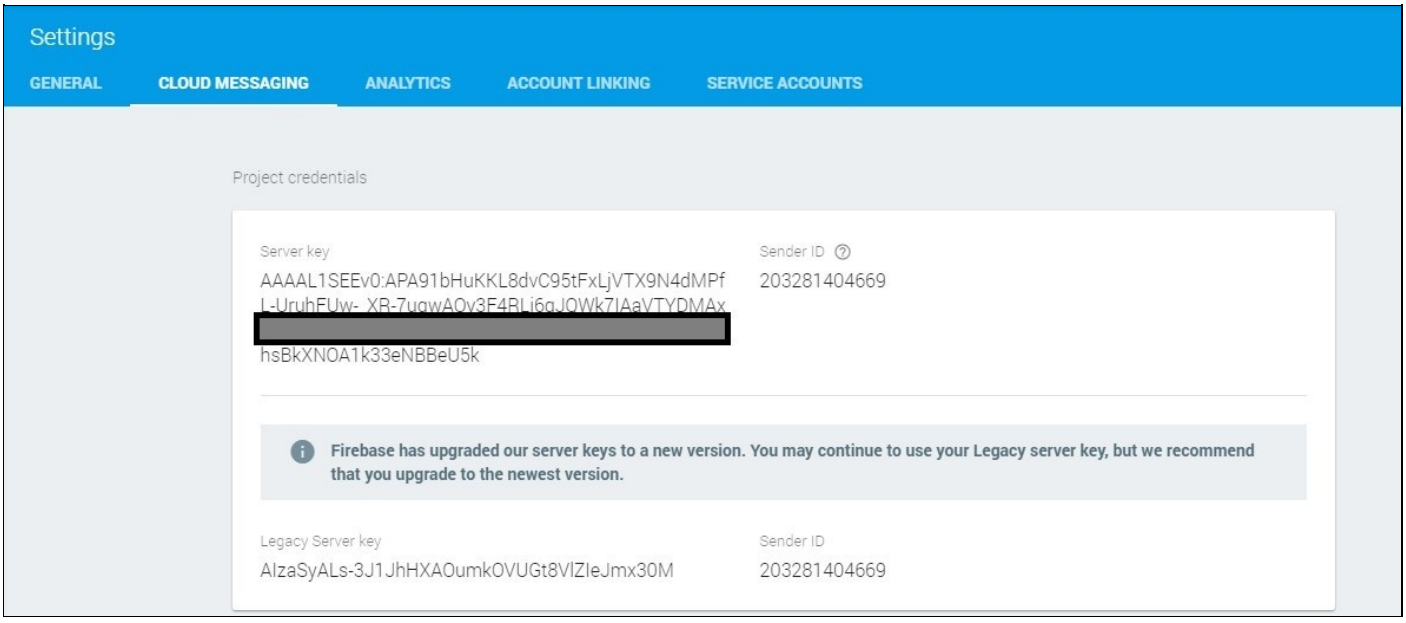
Add the notification feature to the Android Things app

Once you have configured the Firebase project, we can add the notification feature to the alarm system.

Copy the `NotificationManager.java` class shipped with this book's source code into your project under the package `androidthings.project.alarm.util`. This class manages the connection to the Firebase and sends the notification.

Now open `MainActivity.java`and in the `onGpioEdge` method of the `callback` class add the following lines: public boolean onGpioEdge(Gpio gpio) { try {
boolean callBackState = gpio.getValue(); Log.d(TAG,
"Call back state ["+callBackState+"]");
NotificationManager.getInstance()
.sendNotificaton("Alarm!", *server_key*);
}
catch(IOException ioe) { ioe.printStackTrace();
}
return true;
}

Where the `server_key` is the key you get from the Firebase console:



Considering the Android Things app has to use the internet connection to connect to the Firebase cloud service, we have to modify the Manifest.xml requesting the permissions: <uses-permission android:name= "android.permission.INTERNET" />
<uses-permission android:name= "android.permission.ACCESS_NETWORK_STATE" />

That's all; our Android Things app is now ready to send notifications.

Android companion app

In order to receive the notifications, we have to install an Android companion app on our smartphone. Just to simplify the system, this Android app will:

1. Subscribe to the channel used by the Android Things app to send notifications.
2. Implement a service to listen to the incoming notification.
3. Show the notification to the user.

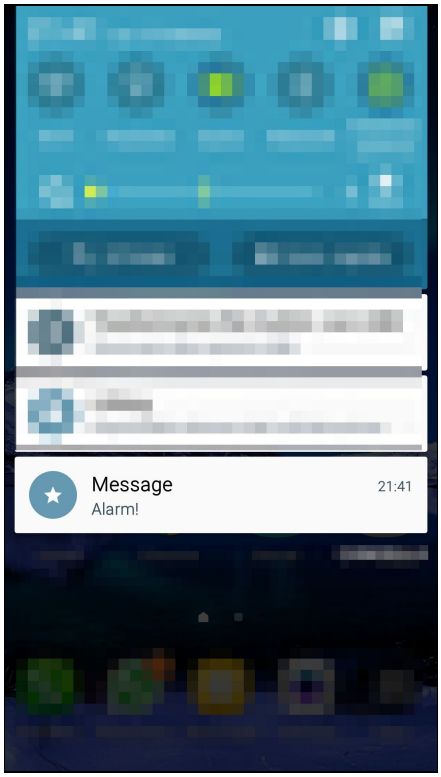
If you do not know how to receive notifications in Android you can visit https://firebase.google.com/docs/android/setup to know more. The source code of the Android companion app is provided with this book's source code. The app interface is very simple because we simply have to subscribe to the topic and wait for the incoming notifications.

To install the app, just open the project using Android Studio and connect your smartphone to your PC/Mac. Add the `google-services.json` file to your app module. This file is the same one that you downloaded in the previous steps. Run and install the app on your smartphone. That's all.

The following screenshot shows the app UI:

Click the SUBSCRIBE button to subscribe your device to the notification channel.

To test the app, you have to install the Android Things app on your board too. As soon as the system detects an object moving, it will contact the Google Firebase platform by sending a notification message. In turn, the Firebase platform will send the message to the user's smartphone. The following is a screenshot of the message received by the user:



(author image issue shows grayscale error)

Below the Android Things app log, notice the body of the message sent to the Firebase cloud platform:

```
androidthings.project.alarm D/MainActivity: Call back state [true] androidthings.project.alarm D/Alm: Send data andr
```

Summary

At the end of this chapter, we implemented an alarm system using Android Things SDK. Moreover, you now know how to use a two-state sensor using GPIO pins. In the last part of the chapter, you learned how to integrate the Android Things app with Google Cloud services such as Firebase to send notifications.

You can use this knowledge to develop this project, adding new features such as more PIR sensors to monitor several rooms at the same time. Moreover, you can use a Firebase real- time database to log the time when the sensor detects movements.

In the next chapter, we will learn how to use more complex sensors that measure physical properties. We will experiment how to use I2C sensors and how we can integrate them with the Android Things app.

How to Make an Environmental Monitoring System

This chapter describes how to build an environmental monitoring system. We want to build a complex IoT system, using Android Things, that measures some physical environment properties. Furthermore, in this Android Things project, we will use RGB LED, introduced in Chapter 1, *Getting Started with Android Things*, and a single color LED to visually represent the environment conditions. To do it, we will use a different class of sensors. While in the previous chapter we learned how to use two-states sensors, in this chapter we will use more complex sensors that require different connections and pins. In more detail, this chapter focuses on learning how to use I2C with Android Things.

Moreover, the main topics covered in this chapter are:

- How to use I2C sensors with Android Things
- How to read data from sensors using Sensor Manager
- How to visualize the data acquired using LEDs
- Overview of I2C protocol
- Custom I2c driver

At the end of this chapter, we will have a full working system that we can use to monitor some physical parameters. We could use it in our homes to detect air properties, or outdoors.

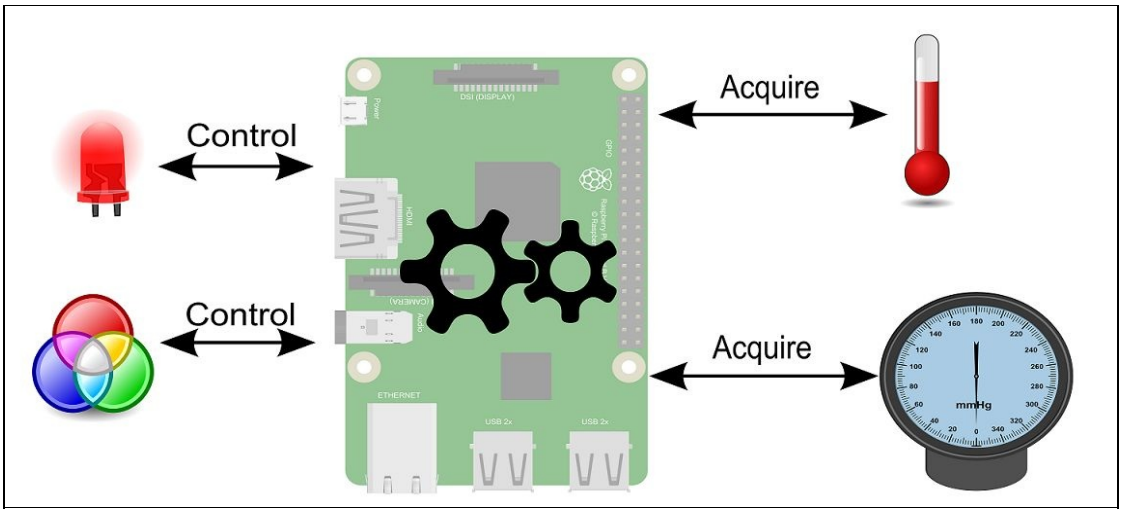
Environmental monitoring system project overview

Before digging into the code details and implementing the project using Android Things, it is useful to have an overview of the project. The target of this project is building an environmental monitoring system that detects:

- Temperature
- Pressure

The interesting aspect of this project, other than the data acquired using sensors, is that the Android Things app will visualize this information using LEDs. In other words, we will implement an app that somehow reacts to the environment properties implementing a custom logic and it is able to control other peripherals.

The following figure shows how the project will work:



This project uses an RGB LED to represent the current pressure condition. The RGB LED will have three different colors:

- **Yellow**: Stable condition. The pressure is over 1022 millibar.
- **Green**: Cloudy. The pressure is between around 1000 millibar and 1021 millibar.
- **Blue**: Chance of rain. The pressure is under 1000 millibar.

There is another red LED that we will use to alert the user when the temperature is lower than a predefined threshold.

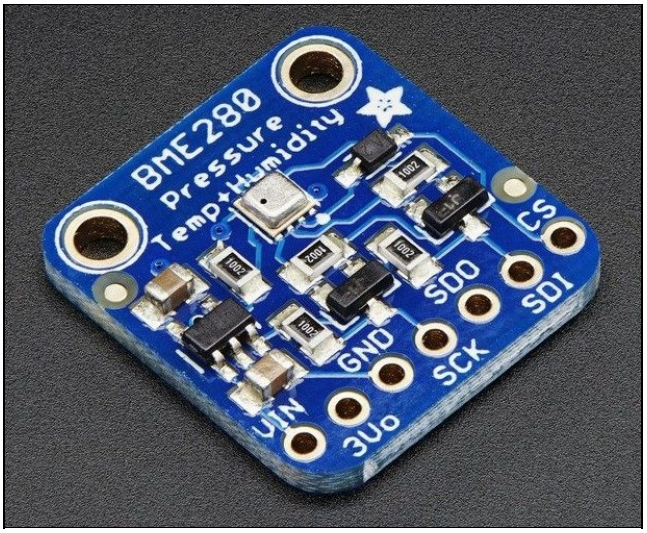
Project components

In order to build this project, the components required are:

Raspberry PI 3 or Intel Edison with Arduino breakout kit

- BME280 or an alternative BMP280
- RGB LED (Common Anode)
- Red LED (Alternatively, you can use another single color LED)
- Resistors (220 Ohm)
- Jumping wires

BME280 is an interesting sensor developed by Bosch that can measure, in only one sensor, all the parameters we want to monitor in this project. This is a low-cost sensor that has a good resolution and it fits perfectly to this project. The following image shows the sensor:



Source: https://www.adafruit.com/products/2652

Instead of BME280, we can use BMP280. This sensor is very similar to the BME280 but it cannot measure the humidity. This a low-cost sensor with comparable features to BME280. The following image shows it:

Source: https://www.adafruit.com/products/2651

Of course, you can use other types of sensors that are compatible with BMP280, or BME280 not only those shown in the preceding images.

Project schematic

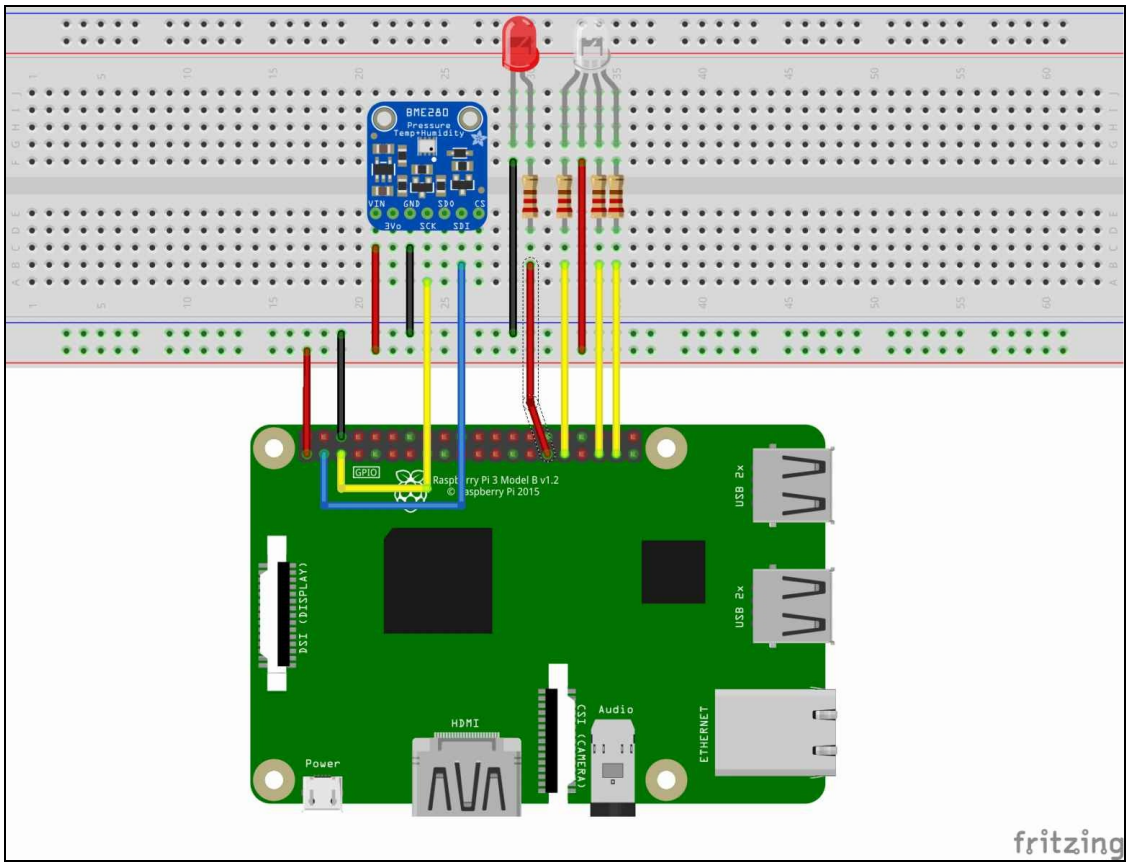
Respects two-state these I2C sensors require more connection toward the Android things board. These types of sensors have several pins; anyway, we are interested in:

- Vin is the power pin. The input voltage must be between 3v-5v.
- GND is the ground pin.
- SCK is the clock signal because I2C sensor uses clock signal as we will see later.
- SDA is the data pin.

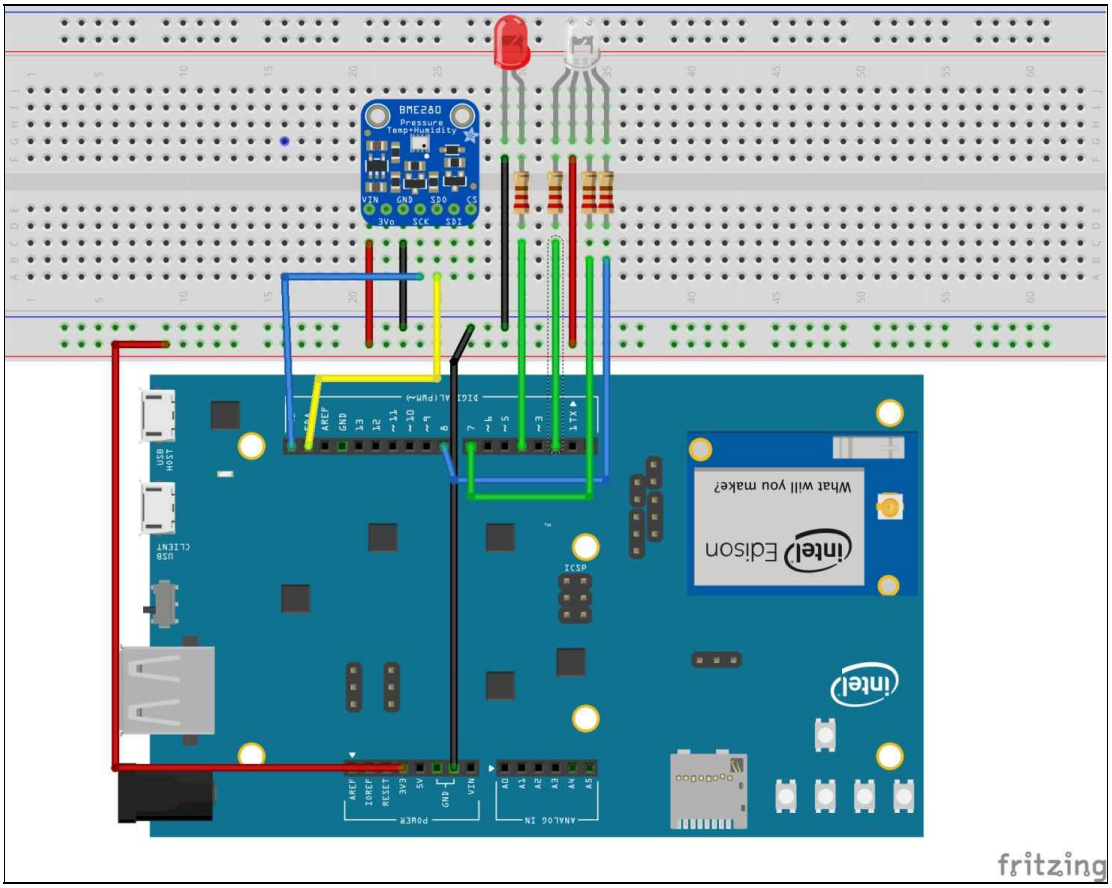
These sensors have other pins, but we will not use them because we will connect them using I2C bus.

Be aware that not all BMP280/BME280 compatible sensors can tolerate a +5V. There are some compatible peripherals, like the one used in this project, that support only +3V. Read the specification before using it in your project.

The following figure shows the sensor connected to Raspberry PI 3:



While if you use Intel Edison with Arduino, the schematic is as follows:

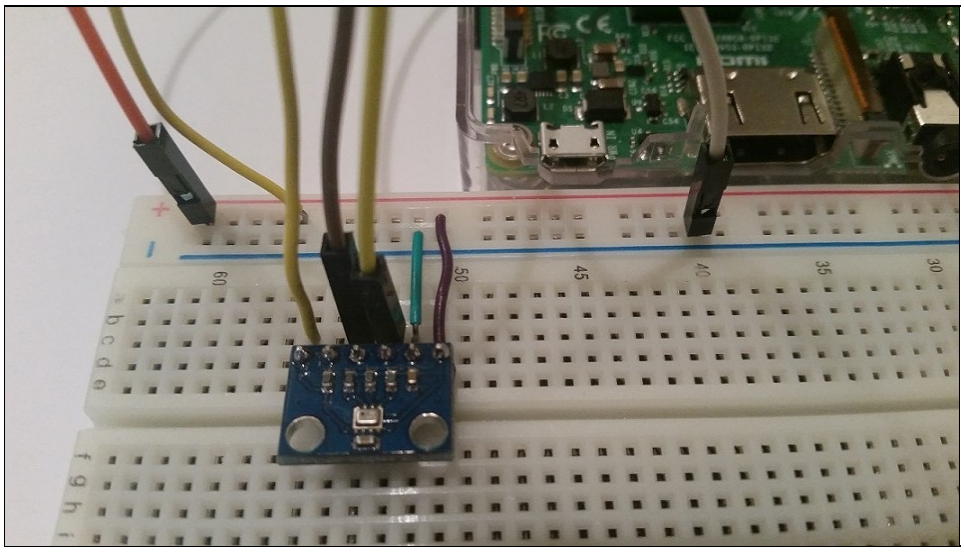


There is an important aspect to consider when connecting the BMP280/BME280. According to its datasheet at https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BMP280-DS001-12.pdf, the SDO pin must be used to select the unique device address. As we will see in more detail later, each peripheral that supports I2C connection has its own address. The address is:

- 0x77 when the SDO pin is connected to the Vcc
- 0x76 when the SDO pin is connected to the ground

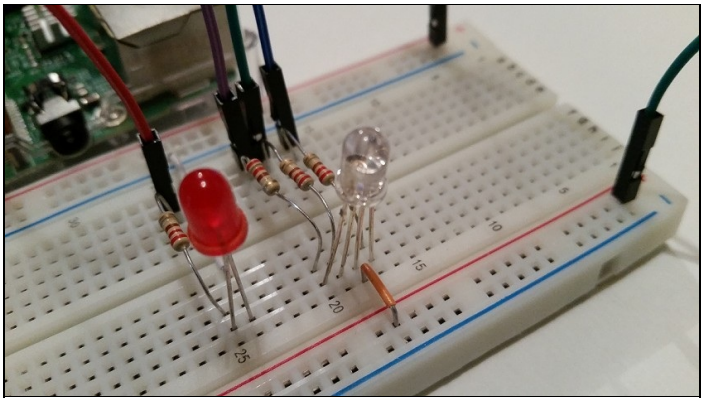
The SDO pin cannot be left floating because the I2C address would be undefined.

Here are some images of the connection details. This first image shows how the BMP280 sensor is connected to the Android Things board:



Notice that the SDO pin is connected to +3V. The SDO pin is the first from the left side.

The following image shows how to connect the RGB LED and the red LED. As you can see, there are 220 Ohm resistors between the LED pins and the Android Things board pins:



Notice that in the RGB LED common anode, the anode pin is connected to +3V.

How to read data from sensors

Now we are ready to start acquiring data from I2C sensors. Usually, in order to use an I2C peripheral, we need a driver. A driver is a set of classes that handle the communication between the Android Things board and the peripheral. Moreover, these classes handle the specific protocols implemented by the peripheral. We will describe how to implement a low-level protocol in the next sections. By now, we can use a pre-built driver that is a library we have to include in our project. All the drivers officially supported by Android Things are available at GitHub under the folder `contrib-drivers` at https://github.com/androidthings/contrib-drivers.

Let us start:

1. Create a new Android Things project by cloning the repository as described in the first chapter.
2. Open `build.gradle` and add the following line under the dependencies:

```
    compile 'com.google.android.things.contrib:driver-bmx280:xx'
```

   Where *xx* is the version of the driver.

   Now you are ready to use the BMP280/BME280 sensor in the project.

3. In `MainActivity.java` in the `onCreate` method add the following lines:

```
        try {
        Bmx280 sensor = new Bmx280(PIN_NAME); sensor.setTemperatureOversampling(Bmx280.OVERSAMPLING_1X); float v
        Log.d(TAG, "Temp ["+val+"]");
        }
        catch(Throwable t) { t.printStackTrace();
        }
```

In the first line, the app instantiates the class that will handle the sensor communication details. In the constructor parameter, this class accepts the *SDA* pin identification. This pin is:

- I2C1 for Raspberry PI 3
- I2C6 for Intel Edison with Arduino breakout kit

In the second line, the app sets the sampling rate. There are different values that control how many samples the sensor will acquire. You can explore it.

Finally, we read the current temperature. In the same way, we can read the pressure:

```
sensor.setPressureOversampling(Bmx280.OVERSAMPLING_1X); float press = sensor.readPressure();
```

When running the Android Things app, the log is shown as follows:

```
02-20 20:03:45.514 5629-5629/? D/MainActivity: onCreate... 02-20 20:03:45.542 5629-5629/? D/MainActivity: Temp [23.1
```

If you run the app again, you may notice that the values read by the sensor are slightly different. This is normal behavior.

Handling sensors using the Android sensor framework

The approach described in the previous chapter works if we want to read the pressure and the temperature one shot. In the project we are developing in this chapter, the app has to read the temperature and the pressure continuously. Therefore, it is convenient to use another approach. This approach is the same strategy we implement in Android when the app needs to monitor the smartphone sensors. As you know, nowadays a smartphone has several built-in sensors and to read their values we use the *sensor framework* provided by Android SDK.

Just to recap briefly how sensor framework works in Android SDK, we can remember that there are key elements that play an important role in these frameworks. These elements are:

- SensorManager
- Sensor
- SensorEvent
- SensorEventlistener

Luckily, these classes and interfaces are also present in Android Things SDK, and they help us to develop smart Android Things apps easily. The following is a brief description of the most important classes and interfaces:

- The `SensorManager` is the base class when we want to deal with sensors. Using `SensorManager` we can register/unregister listeners or list the available sensors.
- The `Sensor` class is the class that represents a sensor and its capabilities.
- The `SensorEvent` class is the class that represents the event triggered by the sensor. An instance of `SensorEvent` holds the following information:
  - The sensor information
  - The data read by the sensor
  - The accuracy
  - The timestamp
- The last class is the `SensorEventListener`. It represents the `callback` class that is invoked when a sensor reads a new value or the accuracy is changed. We will use all these classes in our project because we want to handle the sensor (BMP280/BME280), read its values, and listen to the event triggered when the value coming from the sensor is changing.

Generally speaking, to manage sensors in Android and also in Android Things the steps to follow are:

1. Get an instance of `SensorManager`.
2. Create a `callback` class that implements `SensorEventListener`.
3. Register the `callback` class in order to receive the notifications.

Moreover, in this project, we will use another important class called `SensorManager.DynamicSensorCallback`. This class is useful when we want to receive notifications when a *dynamic sensor* is connected or

disconnected from our board.

Let us see how to implement it in our project:

1. Open `MainActivity.java` again and remove/comment the code used in the previous paragraph.
2. Add the following line:

```
        sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
```

In this way, we obtain the instance of the `SensorManger` from the system sensor manager. The next step is implementing the sensor `callback` class.

Implementing the sensor callback

As described previously, if we want to be notified when the value read by a sensor is changed (or the accuracy is changed) we have to register a listener that extends `SensorEventListener`. In this project, we want to monitor two different parameters:

- Temperature
- Pressure

Therefore, we need two different listeners, one for each sensor. For the temperature sensor the `callback` class is shown as follows:

```
private class TemperatureCallback implements SensorEventListener {
    @Override
    public void onSensorChanged(SensorEvent sensorEvent) {
        float val = sensorEvent.values[0];
        Log.d(TAG, "Temp ["+val+"]");
    }
    @Override
    public void onAccuracyChanged(Sensor sensor, int i) {
        Log.d(TAG, "T. Accuracy ["+i+"]");
    }
}
```

While for the pressure sensor we have:

```
private class PressureCallback implements SensorEventListener {
    @Override
    public void onSensorChanged(SensorEvent sensorEvent) {
        float val = sensorEvent.values[0];
        Log.d(TAG, "Press ["+val+"]");
        }
        @Override
        public void onAccuracyChanged(Sensor sensor, int
           i) { Log.d(TAG, "P. Accuracy ["+i+"]");
    }
}
```

As you can notice from the preceding code, our custom `callback` class overrides two methods:

- `onSensorChanged` that is called when the new value read from the sensor is available
- `onAccuracyChanged` that is called when the accuracy is changed

Moreover, notice that when the value changes the `onSensorChanged` method receives the `SensorEvent`. It represents the event holding all the information we need to identify the sensor that triggered the event and the new value. As you may have already guessed, in the method that handles the new values from sensors we will implement the logic to handle the RGB colors and the red led state (on or off).

How to handle dynamic sensors

Once we have implemented the custom `callback` classes, we have to register them in order to receive the events. We can do it only when the Android Things app is notified that the sensor is connected otherwise we cannot register the listeners. In this step, the app uses `SensorManager.DynamicSensorCallback` to handle this notification event.

In MainActivity.java add the following lines:

```
private class BMX280Callback extends SensorManager.DynamicSensorCallback {
    @Override
    public void onDynamicSensorConnected(Sensor sensor) {
      int sensorType = sensor.getType();
      Log.d(TAG, "On Sensor connected...");
      if (sensorType == Sensor.TYPE_AMBIENT_TEMPERATURE) {
        Log.d(TAG, "Temp sensor..");
        tempCallback= new TemperatureCallback();
        sensorManager.registerListener(
        tempCallback, sensor,
        SensorManager.SENSOR_DELAY_NORMAL);
      }
      else if (sensorType == Sensor.TYPE_PRESSURE) {
      Log.d(TAG, "Pressur sensor.."); pressCallback = new
      PressureCallback(); sensorManager.registerListener(
      pressCallback, sensor,
      SensorManager.SENSOR_DELAY_NORMAL);
    }
  }
  @Override
  public void onDynamicSensorDisconnected(Sensor sensor) {
  super.onDynamicSensorDisconnected(sensor);
  }
}
```

This class apparently seems quite complex, but it does a few things:

1. Override the `onDynamicSensorConnected` method implementing a custom logic:
    1. Get the sensor type.
    2. According to the sensor type, register the sensor `callback` class.
2. Override `onDynamicSensorDisconnected`.

In more detail, in `onDynamicSensorConnected` the app identifies the type of sensor that is connected using:

```
int sensorType = sensor.getType();
```

According to the sensor type (temperature or pressure) it registers the corresponding listener:

```
sensorManager.registerListener(
    tempCallback, sensor,
    SensorManager.SENSOR_DELAY_NORMAL);
```

It is worth mentioning that in the listener, we set the rate used to acquire data or in other words, how fast the `onSensorChanged` method in the sensor listener is called. There are four possible values for the sampling rate:

- `SENSOR_DELAY_NORMAL`: A delay around 200,000 microseconds
- `SENSOR_DELAY_UI`: A delay of 60,000 microseconds

- `SENSOR_DELAY_GAME`: A delay of 20,000 microseconds
- `SENSOR_DELAY_FASTEST`: `0` delay

According to your Android Things app specification and the scenario where the app will work, you have to select the sampling rate that best fits our needs.

Usually, when acquiring environment parameters for this kind of application, the sampling rate should be `SENSOR_DELAY_NORMAL`, because it is not required to acquire data too fast.

Putting it all together - acquiring data

It is time to put everything together and start acquiring data. By now, we have implemented:

- Two sensor listeners to listen to the new values
- The listener to know when the sensor is connected to the board

Let us glue all the pieces and make our app work. Open `MainActivity.java` again and in the `onCreate` method add the following lines:

```
callback = new BMX280Callback(); sensorManager.registerDynamicSensorCallback(callback); try {
  mySensorDriver =
  new Bmx280SensorDriver(BoardPins.getSDAPin());
  mySensorDriver.registerTemperatureSensor();
  mySensorDriver.registerPressureSensor();
}
catch(Throwable t) { t.printStackTrace();
}
```

Where `mySensorDriver` is an instance of `Bmx280SensorDriver` that handles the communication details to the BMP280/BME280. Notice that as we described in the previous chapter, to make the app independent from the board we did not directly use the SDA pin identification, but we have used a method to retrieve the pin name according to the board.

Now we can run the Android Things app and check how the sensor starts acquiring environment parameters. The following is the log of the app:

```
D/MainActivity: On Sensor connected... D/MainActivity: Temp sensor.. D/MainActivity: On Sensor connected... D/MainAc
```

Notice that the app, at the beginning, is notified when the sensor is connected to the board. In this project, the sensor behaves like a double sensor: one that acquires a temperature parameter and another one that acquires a pressure parameter. For this reason, in the app log, there are two calls to the `onDynamicSensorConnected` method. Once all the listeners are configured, the app starts logging the current value of the temperature and of the pressure.

@Override
protected void onDestroy() { super.onDestroy(); Log.d(TAG, "onDestroy");
sensorManager.unregisterListener(tempCallback); sensorManager.unregisterListener(pressCallback); mySensorDriver.unregisterDynamicSensorCallback(callback);
try { mySensorDriver.close();
}
catch (IOException ioe) {}
}

That's all. Now you know how to use an I2C sensor with Android Things. In the next section, we will learn how to use the values read by the sensor to implement a custom logic.

How to control GPIO pins

Now that we know how to read the environment parameters, we can implement the application logic to control other peripherals according to the values acquired. As described in the previous sections, the Android Things monitoring app uses the temperature and pressure to controls two devices:

- An RGB LED that shows the current pressure state
- A RED LED that shows if the temperature is lower than a threshold

To make the app work, we have to fix the pressure threshold values. To simplify the development process we can suppose that there are two thresholds:

- Threshold one, that we will call `LEVEL_1`, is 1022.9 mb
- Threshold two, that we will call `LEVEL_2`, is 1009.14 mb

The app logic that we will implement works in this way:

- If the current pressure is over the `LEVEL_1` then the RGB LED will have the green and red color turned on (yellow)
- If the current pressure is between `LEVEL_1` and `LEVEL_2` the RGB LED will have only the green color turned on
- If the current pressure is below `LEVEL_2` the RGB LED will have only the blue color turned on

Therefore, the RGB LED color can be used to represent the weather forecast:

- If the pressure level is above 1022.9 mb the weather will be stable.
- If the pressure level is between 1009.14 mb and 1022.9 mb then the weather will be cloudy.
- If the pressure level is under 1009.14 mb the weather will be rainy. Of course, this is a really simple weather forecast and we will see later how to improve the project.

The red LED will be used as an alert. The app turns it on when the temperature is under 0°C.

Let us see how to implement it.

Initialize the GPIO pin

The first step is initializing the GPIO pins that the app uses to control the three RGB LED colors and the red LED. As we already learned in the previous chapter, the first step is getting an instance of `PeripheralManagerService`:

1. Open `MainActivity.java` and in the `onCreate` method add:

```
pManager = new PeripheralManagerService();
```

2. Add the following method to the same class:

```
Private void initRGBPins() {
 try
   redPin = pManager.openGpio(BoardPins.getRedPin());
   redPin.setDirection(Gpio.DIRECTION_OUT_INITIALLY_LOW);
   redPin.setActiveType(Gpio.ACTIVE_LOW); greenPin =
      pManager.openGpio(BoardPins.getGreenPin());
   greenPin.setDirection(
      Gpio.DIRECTION_OUT_INITIALLY_LOW);
   greenPin.setActiveType(Gpio.ACTIVE_LOW);
   bluePin = pManager.openGpio(
      BoardPins.getBluePin());
   bluePin.setDirection(
      Gpio.DIRECTION_OUT_INITIALLY_LOW);
   bluePin.setActiveType(Gpio.ACTIVE_LOW);
   redLedPin = pManager.openGpio(
   BoardPins.getRedLedPin()); redLedPin.setDirection(
   Gpio.DIRECTION_OUT_INITIALLY_LOW);
   redLedPin.setActiveType(Gpio.ACTIVE_HIGH);
 }
  catch(IOException ioe) { ioe.printStackTrace();
 }
}
```

This method initializes the pins following these steps:

1. Open the communication to the LED pins.
2. Set the type of the pin. In this case, the app uses the pin in the write mode.

3. Set the pin reference value.

   There is an important aspect to notice in the preceding code. In this project, we are using a common anode RGB LED, so as you may already know, for this kind of LED when the color pin is 0 or low the corresponding color is visible. In other words, it works in the opposite way we would expect. For this reason, the app uses the following line for the blue pin:

```
   bluePin.setActiveType(Gpio.ACTIVE_LOW);
```

The line is repeated for all the RGB LED pins. In this way, the app can set the pin to high or true and the corresponding color turns on. The other lines of code are self-explaining:

4. Add the following line to the onCreate method:

```
initRGBPins();
```

5. Now it is time to implement the real app logic. As you would expect, the app has to change the RGB LED colors when a new value is acquired by the sensor. Therefore, the best place to do it is in the sensor listener methods. For the pressure visualization, we have to modify the pressure sensor listener in the `onSensorChanged` method adding the following lines:

```
int newWeather = -200;
  if (val >= LEVEL_1)
    newWeather = 1;
  else if (val >= LEVEL_2 && val <= LEVEL_1)
    newWeather = 0;
  else
    newWeather = -1;
  if (newWeather != currentWeather) {
     currentWeather = newWeather;
    // Set the RGB color
  switch (newWeather){
    case 1:
      setRGBPins(true, true, false); break;
    case 0:
      setRGBPins(false, true, false); break;
    case -1:
      setRGBPins(false, false, true); break;
   }
}
```

6. In the app, there is a simple trick: to avoid setting the RGB color every time the sensor reads a new value, the app simply checks if the new value could modify the RGB LED color. If the new value implies that the RGB LED has to change color then it calls setRGBPins to change the color. This method is defined as follows:

```
Private void setRGBPins(boolean red, boolean green, boolean blue) {
    try {
    Log.d(TAG, "Change RGB led color. Red ["+red+"] -
      Green ["+green+"] - Blue ["+blue+"]");
    redPin.setValue(red); greenPin.setValue(green);
    bluePin.setValue(blue);
   }
   catch (IOException ioe) { ioe.printStackTrace();
  }
}
```

This method controls the three RGB Led pins.

7. Now we can implement the red LED logic. We want to turn it on when the temperature is under 0°C. Of course, you can set a different threshold. As we did for the pressure, we have to modify the temperature sensor listener, adding the custom logic to handle the red LED state. Look for the `TemperatureCallback` class and in the `onSensorChanged` method add the following lines:

```
boolean turnOn = false;
```

```
          if (val<= 0)
            turnOn = true;
          else
            turnOn = false;
          if (currentState != turnOn) {
            Log.d(TAG, "Change RED led color. New state
            ["+turnOn+"]"); try {
            redLedPin.setValue(turnOn);
            currentState = turnOn;
            }
            catch(IOException ioe) { ioe.printStackTrace();
          }
        }
```

The code is very simple so it does not require any other comment. Now it is time to test the app. You can plug in the Android Things board and run the app from Android Studio. As soon as the installation process completes, the app starts logging as follows:

```
DYNS native SensorManager.getDynamicSensorList return 2 sensors On Sensor connected... Temp sensor.. On Sensor conne
```

compile 'com.google.android.things.contrib:driver-bmx280:xx'

As long as we use peripherals that have a library to handle them we do not have to worry about the protocol details. When we use a peripheral that is not directly supported or there is not a library, we have to implement the specific peripheral protocol. In this context, it is important to know how I2C works.

I2C protocol overview

I2C stands for **Inter Integrated Circuit**. This is a serial communication protocol that uses two wires. I2C is used to exchanging data between integrated circuits. It was developed by Philips in the 80s. During the years, I2C protocol was updated several times and there are different protocols derived from it. One of the most known is SMBUS developed by Intel. Anyway, all these protocols/buses are very similar. The I2C is widely adopted since it is very simple to use. I2C is used to connect low-speed devices such as converters, sensors, and so on. In fact, one of the main drawbacks of this protocol is the speed.

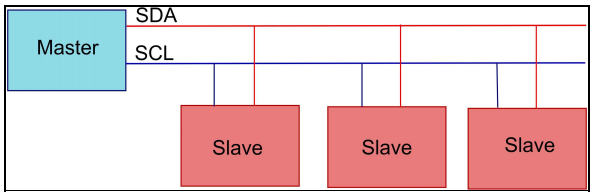
As said before, the I2C uses two wires:

- SCL, that is, the clock
- SDA, that is, the data line

Moreover, this type of bus uses two different nodes:

- A master node that generates the clock signal
- A slave node that uses the clock signal to synchronize its work

Without digging into the details, the most common configuration is a master and one or more slaves connected. Anyway, there are architectures that include more masters and several slaves. The typical configuration, also implemented in this project, is represented in the following figure:



Notice that the ground and the Vcc are missing in the previous diagram for the sake of simplicity.

In the preceding figure, you can recognize the schematic used to connect the sensor to the Android Things board. In the environmental monitoring project, the master is the Android Things board that generates the clock while the sensor is the slave that uses the clock. The SDA line is the one we used to exchange data with the sensor.

Every slave has a unique address that is used to identify it. In order to have an overview of the data flow between master and slaves it is worth mentioning that the communication is initiated by the master and it goes on in this way:

1. The master generates a start condition informing all the slaves that the transmission is going to start.
2. The master sends the slave a unique address with a read (R) or write (W) flag.
3. The slave that has the ID equal to the address sent by the master responds with an ACK signal.

4. The master and slave start exchanging the data.
5. At the end of transmission when all bytes are read or written the master sends the stop signal (P).

When the communication between master and slave ends the bus is free and other slaves and the master can use it to transmit other data. The following is a representation of the data exchanged between the master and the slave:

Using this data packet we can implement a custom library that talks to the sensor. In the following sections, we will describe how to do it using Android Things SDK.

How to implement a custom sensor driver

Now that we know how I2C protocol works, we can start developing our custom driver. A powerful feature of Android Things is the capability to add new peripherals developing specific drivers. In other words, it is possible to extend the sensor framework including new sensors. In this way, Android Things does not make a difference between a built-in sensor and the new sensors. It is possible to handle them in the same way we handle built-in sensors. It is important to notice that the driver that handles the sensor depends on the sensor protocol built on top of I2C bus.

In order to implement a new sensor driver in Android Things we have to follow these steps:

1. Implement a class that extends the `UserSensorDriver`.
2. Describe the sensor specifications and capabilities.
3. Register the sensor driver.

To better understand how these classes are used and the role they play, it is useful to analyze how the BMP280/BME280 sensor driver is implemented. This can help us to better understand how user sensors work.

To this purpose, we can clone the repository that contains the official drivers supported by Android Things:

1. Open your browser and go to https://github.com/androidthings/contrib-drivers.

2. You will get a page like the following:

3. Click on Clone or download to copy the repository locally.
4. Now look for the `bmx280` folder and navigate it until you find two classes called:

```
Bmx280.java
BmxSensorDriver.java
```

These two classes are those that manage the BMP280/BME280. Opening `Bmx280SensorDriver.java` you will notice that there are two inner classes called:

- `PressureUserDriver`

- `TemperatureUserDriver`

Both classes extend `UserSensorDriver` and they are the classes that handle the sensor. In this case, we have two drivers because we are measuring two properties. Moreover, both classes define the user sensor properties according to the sensor specifications derived from the manufacturer datasheet. The following code is the sensor definition for the temperature driver:

```
mUserSensor = UserSensor.builder()
.setType(Sensor.TYPE_AMBIENT_TEMPERATURE)
.setName(DRIVER_NAME)
.setVendor(DRIVER_VENDOR)
.setVersion(DRIVER_VERSION)
.setMaxRange(DRIVER_MAX_RANGE)
.setResolution(DRIVER_RESOLUTION)
.setPower(DRIVER_POWER)
.setMinDelay(DRIVER_MIN_DELAY_US)
.setRequiredPermission(DRIVER_REQUIRED_PERMISSION)
.setMaxDelay(DRIVER_MAX_DELAY_US)
.setUuid(UUID.randomUUID())
.setDriver(this)
.build();
```

Notice the several parameters that we have to provide in order to describe the sensor. It is important that just before the `build()` method, we have to attach the sensor driver to the sensor (`setDriver()` method)

So far, we have used the BMP280 to read the temperature and the pressure. We already know that BME280 can read the humidity too. What if we want to extend this driver to add this new feature? Surely, we have to define another class that extends `UserSensorDriver` to handle the humidity property. We simply have to copy and paste the `TemperatureUserDriver` and change the class name in `HumidityUserDriver`. Do not forget to change the sensor definition, as shown previously.

It is worth noticing that the protocol details are not handled in the user driver class, but in another class called `Bmx280.java`. This is the class that communicates directly with the sensor and it is important to know how it works.

Low-level sensor driver

The `Bmx280.java` is the class that makes the heavy work and if we want to implement the new feature that reads the humidity we have to modify it. Moreover, you need the manufacturer datasheet (https://cdn-shop.adafruit.com/datasheets/BST-BME280_DS001-10.pdf) in order to know how to exchange data with the sensor. Let us examine it.

To open a connection to an external sensor we need an instance of the `PeripheralManagerService` class:

```
PeripheralManagerService pioService = new PeripheralManagerService();
```

Then, the class opens the connection using:

```
I2cDevice device = pioService.openI2cDevice(bus, I2C_ADDRESS);
```

Where:

- The bus is the SDA pin we have used to connect the sensor to the Android Things board.
- The `I2C_ADDRESS` is the unique ID of the sensor. Do not forget that the protocol we are using is the I2C and each sensor has a unique address as described in the previous paragraph.

The code line shown previously introduces another interesting aspect. Using `PeripheralManagerService` we can open not only the GPIO connection, as described in the previous chapter, but also the I2C bus connection. In this case, we get an instance of I2cDevice that represents the communication bus where we can read and write data.

Generally speaking, an I2C sensor has a set of registries that we can use to:

- Read data
- Write data

The registries where we can write data are useful to set the sensor behavior. We will use some of them later to activate new sensor features. Moreover, it is important to keep in mind that each registry has a length expressed as a number of bits. We have to know the registry length in order to know how many bits we have to read or write. The registries and their length are described in the sensor datasheet.

Now, look for the connect method. The first line in this method reads the sensor type:

```
mChipId = mDevice.readRegByte(BMP280_REG_ID); // 0xD0
```

According to the sensor specification, the registry that holds the chip information is 0xD0. This registry is 8bit or 1 byte. For this reason, the app uses the `readRegByte` method. After it, the driver reads the registries that hold the temperature and pressure calibration parameters. These parameters are useful to calibrate the temperature and the pressure values read by the sensor. To add the humidity feature to this driver we have to also read the humidity calibration parameters. The following table shows the registry addresses and their length:

Registry address	Registry content	Data type
0xA1	dig_H1 [7:0]	unsigned char
0xE1 / 0xE2	dig_H2 [7:0] / [15:8]	signed short
0xE3	dig_H3 [7:0]	unsigned char
0xE4 / 0xE5[3:0]	dig_H4 [11:4] / [3:0]	signed short
0xE5[7:4] / 0xE6	dig_H5 [3:0] / [11:4]	signed short
0xE7	dig_H6	signed char

This table is extracted from the sensor datasheet. Knowing the registries to read we can develop a few lines of code to get their values. Look for the `connect()` method and add the following lines:

```
int dig_H1 = ((byte)device.readRegByte(0xA1) & 0xFF); byte[] buffer = new byte[7]; mDevice.readRegBuffer(0xE1, buffe
int dig_H2 = (buffer[0] & 0xFF) + (buffer[1] * 256); // 0xE1 0xE2 int dig_H3 = buffer[2] & 0xFF ; // 0xE3
int dig_H4 = ((buffer[3] & 0xFF) * 16) + (buffer[4] & 0xF); // 0xE4 0xE5 (first 3)
int dig_H5 = ((buffer[4] & 0xFF) / 16) + ((buffer[5] & 0xFF) * 16); // 0xE5 (7:4) 0xE6
int dig_H6 = buffer[6] & 0xFF; // 0xE7
```

Congratulations! You know how to read the humidity calibration parameters that will be useful when you read the humidity value. There is another important step that we have to do before using the sensor: we have to enable the humidity. This feature is disabled by default.

To enable it, we have to write the registry at location 0xF2 setting the oversampling value. We can do it in the method that sets the temperature oversampling. Look for the `setTemperatureOversampling` and add the following line:

```
// Enable Humidity sensor mDevice.writeRegByte(0xF2, (byte) 0x1);
```

So the method becomes:

```
public void setTemperatureOversampling(
    @Oversampling int oversampling) throws IOException {
    if (mDevice == null) {
      throw new
      IllegalStateException("I2C device not open");
    }
    // Enable Humidity sensor mDevice.writeRegByte(0xF2,
    (byte) 0x1);
    int regCtrl = mDevice.readRegByte(BMP280_REG_CTRL)
    & 0xff;
    if (oversampling == OVERSAMPLING_SKIPPED) {
    regCtrl &= ~BMP280_OVERSAMPLING_TEMPERATURE_MASK;
    } else {
```

```
    regCtrl |= 1 <<
    BMP280_OVERSAMPLING_TEMPERATURE_BITSHIFT;
    }
    mDevice.writeRegByte(BMP280_REG_CTRL, (byte)
    (regCtrl)); mTemperatureOversampling = oversampling;
    }
```

That's all. Now we can run the app, directly invoking the sensor as we did at the beginning of this chapter. The result is shown as follows:

```
Driver: Connecting... Driver: Sensor type [96] Driver: ---- Humidity calibration parameters ---- Driver: H1 [75] - H
```

There are some interesting aspects to note:

- The sensor type is 96 (0x60). This value means that we are using a `BME280` class.
- The calibration parameters are all available.
- The humidity data acquisition is enabled: its value is 1.

Now we can read the humidity value. According to the sensor datasheet, the sensor stores this value in the registry starting from 0xFD to 0xFE. Therefore, we have to read 16 bits or 2 bytes. Add the following method to the `Bmx280` class:

```
public long readHumidity() throws IOException {
  byte[] dataBuffer = new byte[2];
       mDevice.readRegBuffer(0xFD, dataBuffer, 2);
  long value = (dataBuffer[0] & 0xFF) * 256 +
       (dataBuffer[1] & 0xFF);
  return value;
}
```

This method returns the value read by the sensor. This is a long value and it does not represent the real humidity. It is necessary to convert it using the compensation parameters we have retrieved before. Let us add another method to the same class:

```
public double readCompansatedHumidity() throws IOException {
  long adH = readHumidity();
  float temp = readTemperature();
  double var_H = temp - 76800.0;
     var_H = (adH - (dig_H4 * 64.0 + dig_H5 / 16384.0 *
     var_H)) * (dig_H2 / 65536.0 * (1.0 + dig_H6 /
     67108864.0 * var_H * (1.0 + dig_H3 / 67108864.0
     * var_H)));
  double humidity =
     var_H * (1.0 -dig_H1 * var_H / 524288.0);
  return var_H;
}
```

This compensation formula is described in the sensor datasheet. Now we are ready to read the humidity using this simple test class:

```
try {
  androidthings.project.weather.Bmx280 sensor = new
  androidthings.project.weather.Bmx280(
     BoardPins.getSDAPin());
  sensor.setTemperatureOversampling(
     androidthings.project.weather.Bmx280.OVERSAMPLING_
       1X);
  sensor.setPressureOversampling(
     androidthings.project.weather.Bmx280.OVERSAMPLING_
       1X);
  long adH = sensor.readHumidity();
  double hum = sensor.readCompansatedHumidity();
  Log.d("App", "ADH ["+adH+"]");
```

```
  Log.d("App", "Hum ["+hum+"]");
  }
  catch(IOException ioe) { ioe.printStackTrace();
}
```

Finally, running the app, we have the result we were looking for:

```
Driver: Connecting... Driver: Sensor type [96] Driver: ---- Humidity calibration parameters ---- Driver: H1 [75] - H
App: ADH [32768] App: Hum [69.1864670499461]
```

That's all; now you can master I2C sensors!

Summary

At the end of this chapter, you learned how to use I2C sensors and how to connect them to an Android Things board. Moreover, we implemented a full-working Android Things IoT app that monitors environmental parameters. We have learned how to control GPIO pins using the information retrieved from sensors. This project can be extended by adding new features. An interesting feature that you could add is considering if the pressure is rising or lowering to have a more detailed weather forecast. Using this environmental monitoring system, you gained knowledge about how to implement custom drivers. In this way, you have infinite possibilities to use your Android Things board with several I2C sensors.

In the next chapter, we will cover an important aspect of the IoT ecosystem: IoT cloud platforms. We will learn how to use IoT platforms and how to integrate them with Android Things and stream data to the cloud.

Integrate Android Things with IoT Cloud Platforms

In this chapter, we will learn how to integrate Android Things with IoT cloud platforms. This is an important aspect when developing an IoT app. As a matter of fact, there are several scenarios where it is required that the data acquired from Android Things boards must be transferred to the cloud. For this reason, this chapter will give you all the information you need to connect your Android Things board to IoT cloud platforms.

In this chapter, we will look at the following topics:

- IoT cloud architecture
- How to configure an IoT cloud platform
- How to connect an Android Things app to the IoT cloud platform
- How to stream real-time data to the cloud and create dashboards

During this chapter, we will reuse our Android expertise to handle HTTP communication.

IoT cloud architecture

By now we have explored how to develop Android Things apps that are self-contained. In other words, we have built Android Things apps that do not communicate with external systems or platforms. The data acquired through the sensors are managed locally. There are other scenarios where the Android Things app sends the data acquired to the cloud so that this information is analyzed and integrated with other kinds of data producing new services. In this scenario, the IoT platforms play an important role. Before digging into the IoT cloud architecture details and describing how to integrate Android Things with these platforms, it is important to clarify what we mean by the IoT cloud platform.

An IoT cloud platform overview

Nowadays, the IoT cloud platforms are an important brick in IoT ecosystem. Using these platforms, we can extend the services we can provide and unlock the power of Android Things boards. Through IoT platforms, it is possible to give to the data acquired new meanings integrating it with other information sources. Even if the Android Things board is very powerful, they cannot provide services where big computation power is required. Transferring the information acquired to the IoT cloud platform we move, at the same time, some piece of the business logic from the Android Things to the cloud. Once the data is available at IoT cloud level, these platforms can apply complex analysis and, as a result, they can remotely control the Android Things boards. In this reference context, you can think about all the technologies related to machine learning, **Artificial Intelligence** (**AI**) and big data analysis. The Android Things boards may not have the computation power required by these technologies, but, at the same time, the services developed using these technologies need the data acquired and managed by the Android Things.

What kind of services can IoT cloud platforms provide? There are several services and every IoT cloud platform has its unique features. Generally speaking, these services can be grouped into these categories:

- Connection service
- Data storage service
- Event processing service
- Device management
- Data visualization
- Service integration

All these services are very useful when we are developing an IoT Android Things app. Almost all the IoT platforms provide a *connection service*. The core of these services is the connection and the data transfer between the IoT cloud platform and the remote board. They support different protocols to simplify the connection process. The most common are:

- Rest API and HTTP
- MQTT
- CoAP

In other words, they provide a set of software interfaces that can be invoked by remote IoT boards to connect and exchange data. Moreover, they provide a set of SDKs for different boards to make the connection process easier and fast.

Data storage service is the capability of an IoT cloud platform to store data. Usually, this service is useful when we want to store the data acquired somewhere outside of the Android Things board. This information is the base for other services.

The first two services are usually provided by almost all IoT platforms, while the event processing is a more complex service. This is a rule-based engine that uses the data stored and events to trigger actions that could have effects on the IoT boards. A simple example is the temperature monitoring system that

triggers an alert when a value moves outside a prefixed range. Another example is sending a message to the IoT board to turn on a water pump when the soil moisture is lower than a threshold. Usually, all these events and actions are configured via web interface.

A device management service takes care of managing all the IoT devices connected to the platform. This means, for example, updating the device firmware remotely, changing configuration parameters, and so on. In other words, it is a centralized administration console for remote devices.

Data visualization is the service provided by several IoT cloud platforms to create dashboards to graphically visualize the data acquired using charts.

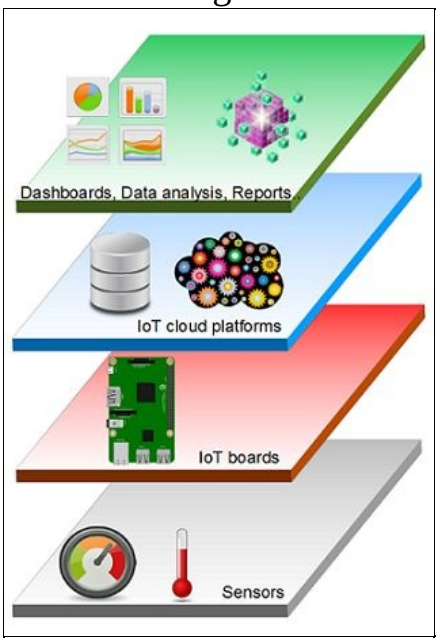
The last service is the integration service. This type of service is useful when we want to integrate some external services and trigger them according to preconfigured events. To have an idea about these external services, think about sending email messages, sending Twitter messages, invoking remote services, and so on. We will cover it later in this chapter.

There are several IoT platforms available on the market. They offer one or more of the mentioned services and everyone has its own features that make it different from others. We have to choose the right platform according to our needs and scenarios where we will use our Android Things app. Here are a few of these IoT cloud platforms:

- Google IoT cloud
- Microsoft Azure IoT
- Amazon AWS IoT
- Samsung Artik Cloud
- Temboo
- Ubidots

IoT cloud architecture overview

Now we know the services the IoT cloud platforms provide, we can define an IoT cloud architecture and the roles that Android Things and IoT cloud platforms play. The following figure describes a possible architecture:



This layered architecture describes the role the components play and how they are placed:

- At the lowest level of this architecture, there is the sensor layer. This layer is where we start acquiring data.
- The second layer is the IoT boards; in this book for IoT boards we mean Android Things compatible boards such as Raspberry PI 3 and Intel Edison. Anyway, there are other IoT boards that are not compatible with Android Things OS, but they can send data to the cloud.
- The third layer is the IoT cloud platform with the services we described previously. An IoT cloud platform collects data coming from sensors through IoT boards and stores it somewhere. Moreover, it can apply one or more services to analyze the data, transform it, and integrate it with other sources. It can use such information to fuel complex engines (AI, machine learning, predictive analytics, and so on).
- The fourth layer is the last layer and it represents the high-level services that are exposed to the final user. They can be, for example, dashboards to visualize information or the results of complex services.

Sometimes, some services at the third and fourth layer are mixed together.

Streaming data to the IoT cloud platform

Once we know what an IoT cloud platform is and the reference architecture, we can implement an Android Things app that streams real-time data to the cloud. Generally speaking, to use an IoT cloud platform we have to follow these steps:

1. Configure the IoT project on the cloud platform providing all the information including the type of the data we want to manage.
2. Create an IoT platform client on the client side (Android Things app) that handles the connection and sends the data.

In this Android Things IoT project, we will use Samsung Artik Cloud (https://artik.cloud/) as the IoT cloud platform,. This is a professional platform that provides almost all the services described in the previous paragraphs. Moreover, it is easy to use and it provides several SDKs that simplify the data exchange process. In this project, we will manually implement the data exchange between the Android Things board and Samsung Artik Cloud so that we can learn all the steps to follow and how to implement them in Android Things. To integrate our Android Things app with Artik Cloud we will use the Rest APIs provided by Artik Cloud itself.

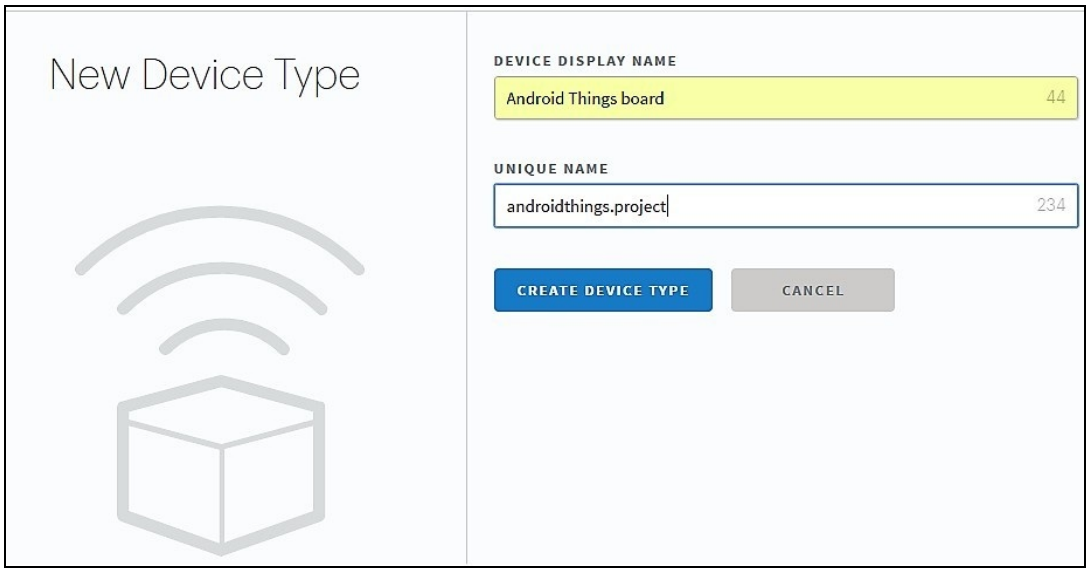
How to configure Artik Cloud

As stated before, the first step is configuring the IoT project on Artik Cloud. To start, it is necessary to create a free account using this link https://accounts.artik.cloud/signup?client_id=d18f11efb5244c8f99f1ac7aa4fb9bbc&redirect_uri=https://my.artik.cloud/authorize. The aim of the configuration steps is creating the `Manifest` file that represents the data model we will use.

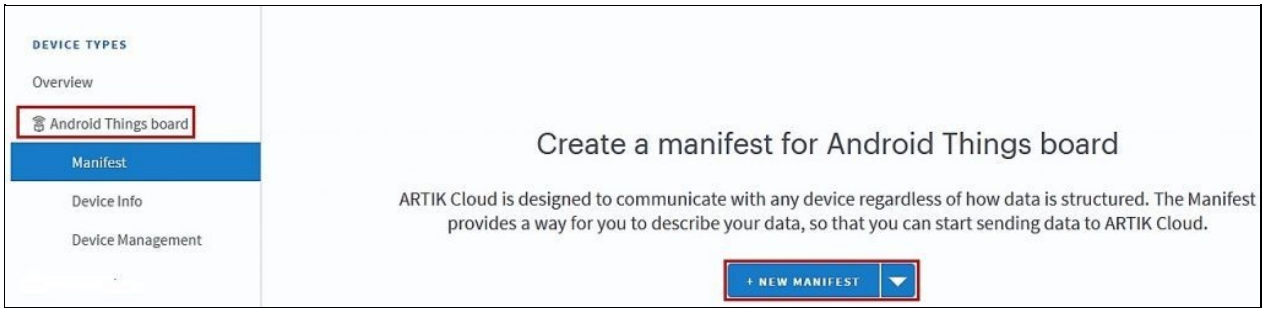
Be aware that the Manifest file of Artik Cloud is not related to the Android Things `Manifest.xml`*. It is a completely different file.*

To configure the IoT project in Artik Cloud, we have to follow these steps:

1. Go to the Artik Cloud developers page (https://developer.artik.cloud/).
2. Click on Device Type. A device type is an abstract representation of our device. Each device type is related to a `Manifest`.
3. Click on New Device Type to add our custom device representation. We have to provide the display name that is a name we like and the UNIQUE NAME, as shown in the following screenshot:



4. Once you have configured a new device type, we can create the Manifest by clicking on NEW MANIFEST, as shown in the following screenshot:

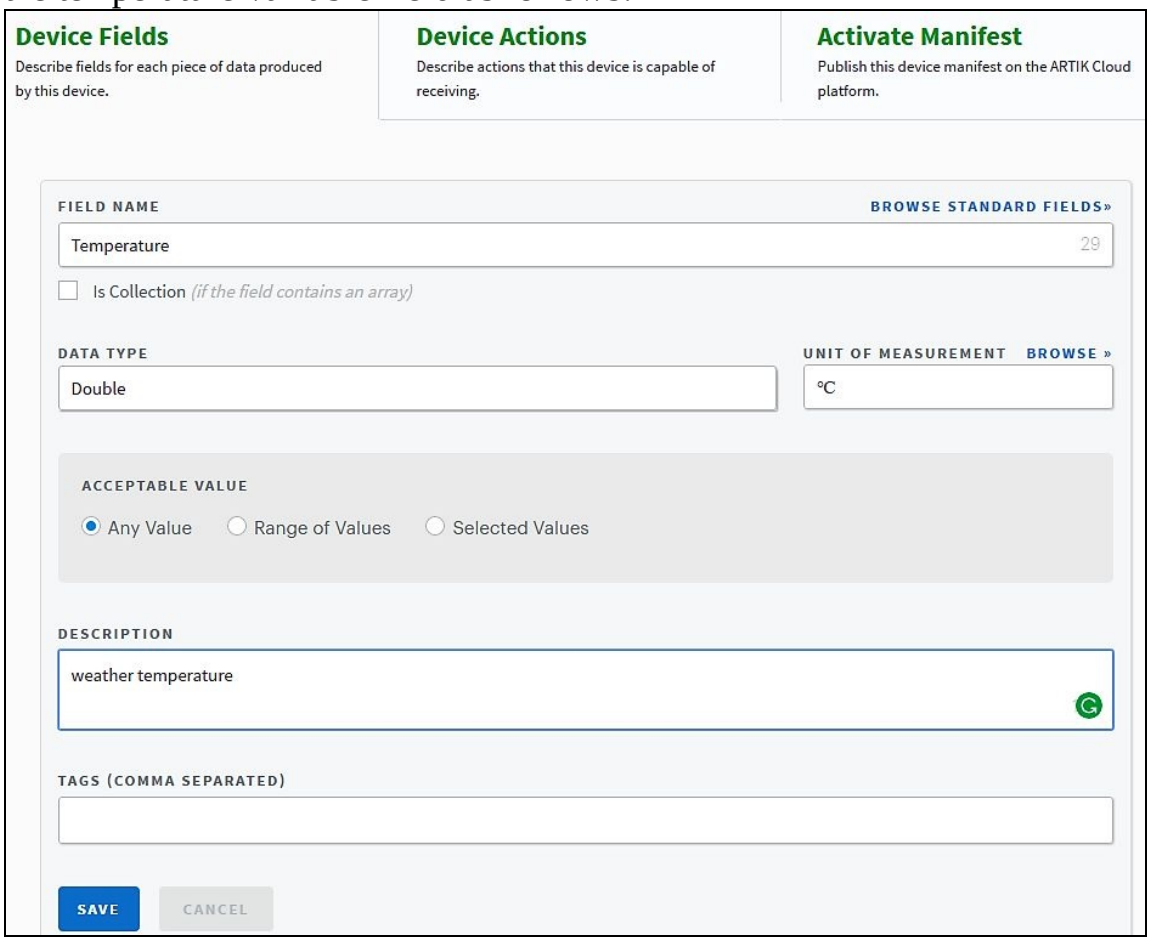


5. Artik Cloud supports several platforms and it is data agnostic. In other words, clients can send different data structures to Artik Cloud. In turn, it uses the *Manifest* to interpret such data. In other

words, when we configure the *Manifest*, we inform Artik Cloud about the data our app will send so that it can retrieve the values sent by the Android Things app. In this project, we want to use two parameters. They are the environment variables that the app acquires from sensors:

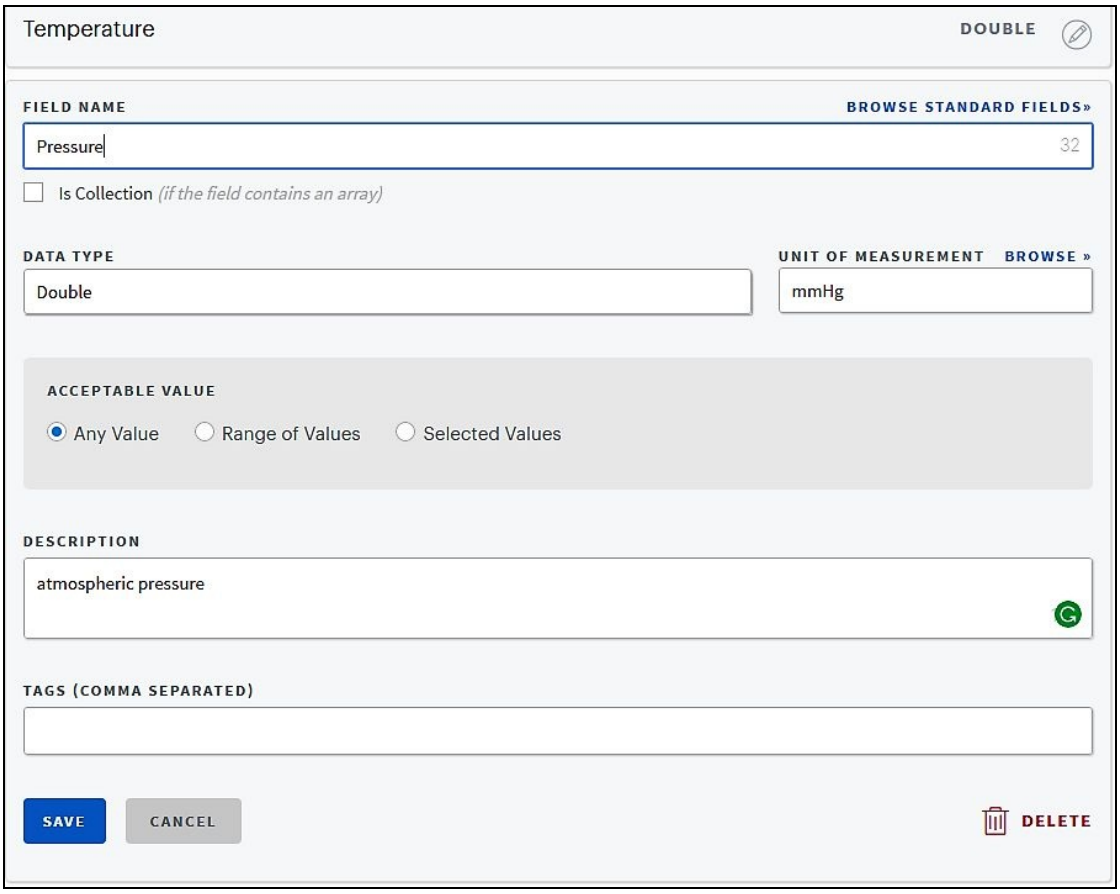
- Temperature
- Pressure

Therefore, we have to configure two variables in the Manifest. In this first step, we configure the temperature variable field as follows:

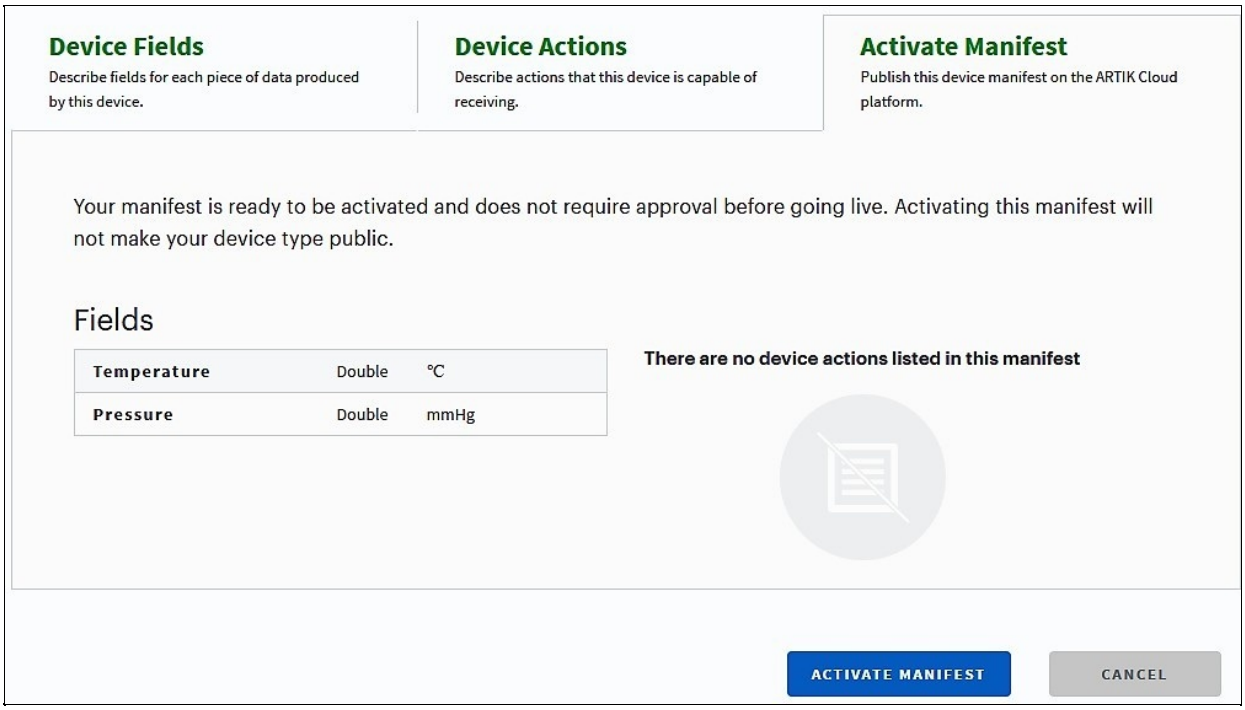


There are a few required fields necessary to configure a new variable. The most important is the name, which we will use later to reference this variable.

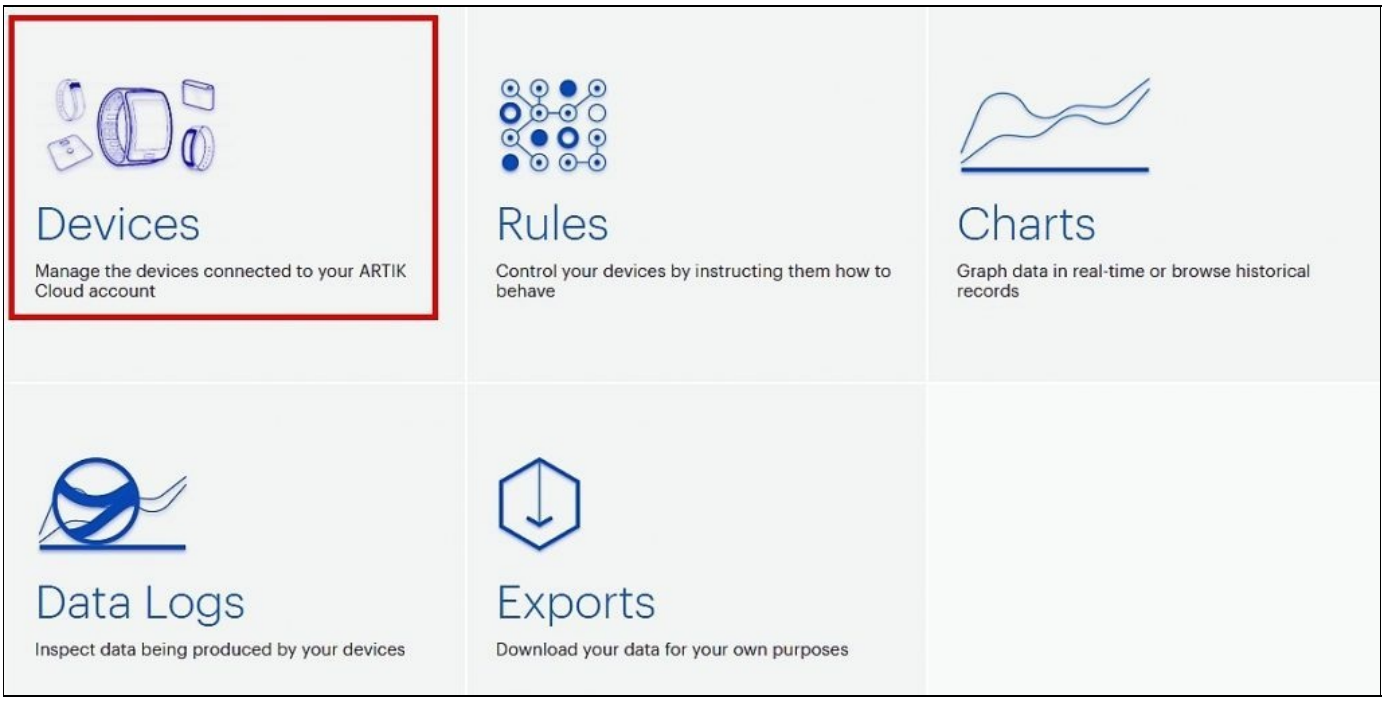
6. Click on NEW FIELD to add the `Pressure` variable:



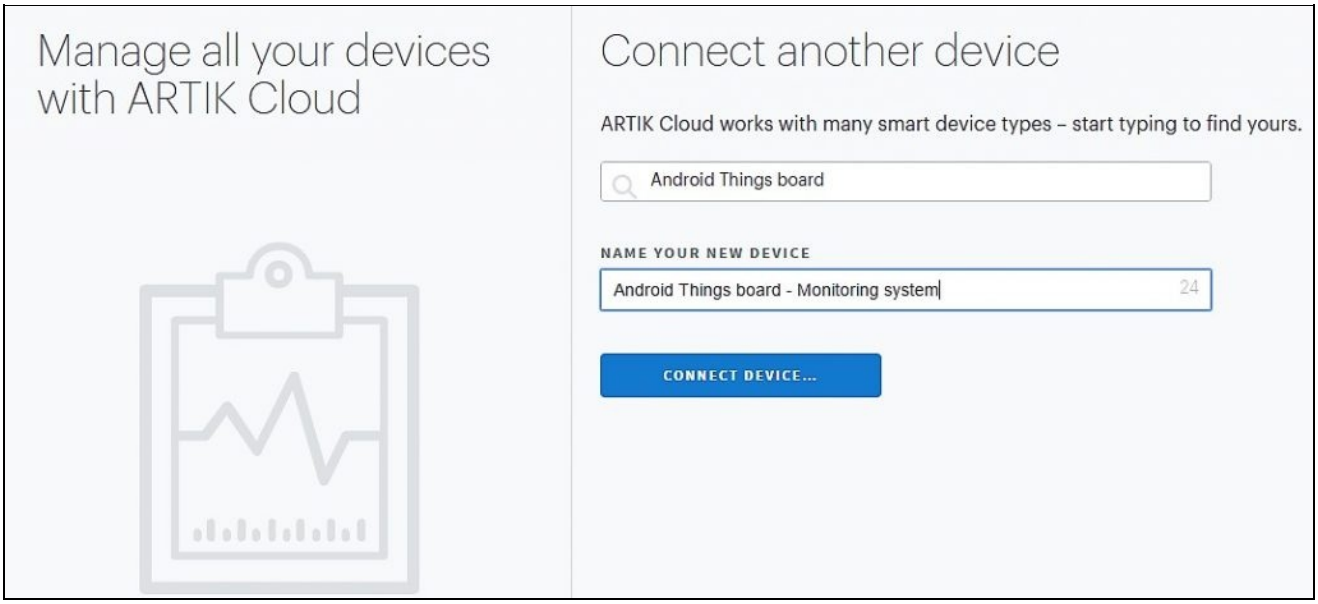
7. Now we can activate the `Manifest` we have configured, by clicking on ACTIVATE MANIFEST:



8. Now click on My Artik Cloud on the top of the screen to move to your dashboard.
9. We are going to create a new device, so click on Devices. A device is a new instance of the device type that we have created before:

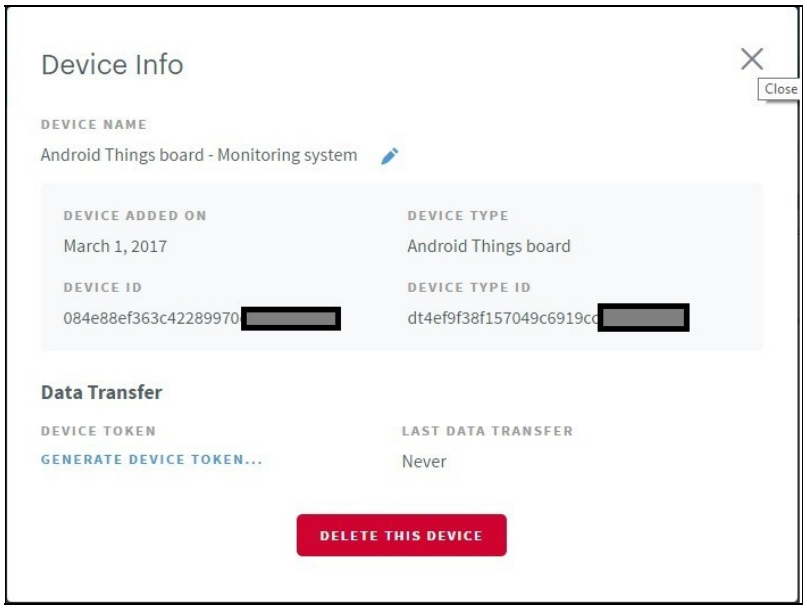


10. Now we can configure the device that represents our Android Things board. It is important that you select Android Things board in the first field:



As the name of the device, we will use *Android Things board - Monitoring system*.

11. Once we have created the device, click on the device details. In this way, we retrieve all the information we need to authenticate our device and send data to Artik Cloud. We will use this information later when we will develop the Android Things client. The following screenshot shows the device detail information:



That's all. You have now configured our IoT project in Artik Cloud and we are ready to send data from our Android Things app.

Artik client description

Once the IoT project and its definition are configured on Artik Cloud, we can analyze how to connect a client to Artik. As stated previously, the target of this project is sending the environmental parameters to the cloud so that we can log them and create charts. To this purpose, we have to modify the Android Things app we developed in the previous chapter and we have to add cloud capabilities. Before modifying the Android Things app, it is important to know the steps we have to follow to connect a client to Artik Cloud:

1. Connect to the Artik Cloud and handle the HTTP connection.
2. Authenticate the device.
3. Invoke Artik Cloud Rest API to send data.

According to the Artik Cloud documentation, the URL to invoke to send data is https://api.artik.cloud/v1.1/messages. In more detail, we have to invoke this URL passing:

- The information necessary to authenticate our Android Things client
- The message. This will hold the data (values acquired from sensors) and other information

In order to authenticate the client the HTTP request header must contain this parameter:

```
Authorization: Bearer device_token
```

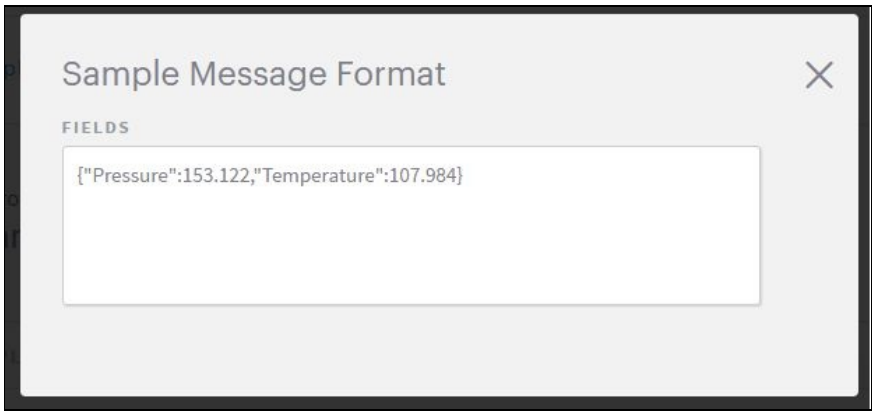
To get the `device_token` we use the Artik Cloud web interface:

1. Go to the Artik cloud dashboard as described previously.
2. Click on devices and then on the Android Things board - monitoring system.
3. In the popup, click on GENERATE DEVICE TOKEN.

Moreover, the message the client sends to the Artik cloud must have a specific structure. Using the Artik Cloud web interface we can obtain an example of the data structure we have to send:

1. Go to the developer console https://developer.artik.cloud/.
2. Click on Device Type and then on Android Things board.
3. Click on the *Manifest* menu item and you should get something like the following:

4. Now click on **VIEW SAMPLE MESSAGE** to know the data structure:



Furthermore, the data containing the sensor values must be contained inside a *message*, which is a wrapper that adds other information to the real data. The following is the message structure:

```
{
      "sdid":"device_id",
      "ts": timestamp,
      "data":
    {
      "Pressure":153.122,
      "Temperature":107.984
     }
}
```

Here, `device_id` is the unique device identification and *ts* is the timestamp. Now we know how to build the message and how to retrieve the authentication parameters.

Therefore, we can implement the Android Things client.

How to implement the Android Things Artik client

To this purpose, we will reuse the Android Things app developed in the previous chapter. The Android Things app has to handle the HTTP connections and to do it we have two options:

- Use Android HTTP library
- Use a custom library

Do not forget that an Android Things app is an Android app so we can reuse the Android HTTP libraries already available. In this project, we use Volley (https://developer.android.com/training/volley/index.html). This library is widely used and offers interesting features. Moreover, it simplifies the HTTP connection management.

To use Volley, follow these steps:

1. Open the `build.gradle` file and add the following lines:

```
dependencies {
             ...
             compile 'com.android.volley:volley:1.0.0'
        }
```

In this way, we are adding the dependency from the volley library.

2. The next step is requesting the permission to connect to the internet. We do it in the `Manifest.xml` file. Add the following line:

```
<uses-permission android:name="android.permission.INTERNET" />
```

3. Now let us create a new class that will handle all the communication details. We will call it `ArtikClient.java`.
4. The `ArtikClient` is a *singleton;* therefore we have to create a private constructor:

```
private ArtikClient(Context ctx, String deviceId, String token) {
            this.ctx = ctx;
            this.deviceId = deviceId;
            this.token = token;
        createQueue();
        }
```

The constructor accepts as parameters the `Android Context`, the `device id`, and the `token`, as described in the previous paragraphs.

5. Implement the `createQueue()` method. This method initializes the `Volley` request queue so that the app can make requests to the Artik Rest APIs:

```
private void createQueue() {
           if (queue == null)
        queue = Volley.newRequestQueue(
                ctx.getApplicationContext());
        }
```

Here, the queue is defined as follows:

```
private RequestQueue queue;
```

6. Once we have configured the request queue, we can focus on implementing the method that sends data. To this purpose, add the `sendData()` method to this class. This is the heart of the class because it invokes the Artik Rest APIs using the message structure described previously. Before writing this method, it is worthwhile to split it into several steps, because it can improve its readability. The steps are:
    1. Create an instance of `StringRequest` that represents the request we want to send.
    2. Override the `getHeaders()method` to customize the HTTP request headers.
    3. Override the `getBody()method` to customize the body we are sending.

The following is a detailed description of the three preceding steps.

Implement a StringRequest with Volley

The `StringRequest` in `Volley` represents the HTTP request we send to Artik. Add the following lines to the `sendData()` method:

```
StringRequest request = new StringRequest(Request.Method.POST,
    ARTIK_URL,
    new Response.Listener<String>() {
      @Override
      public void onResponse(String response) {
        Log.d(TAG, "Response ["+response+"]");
      }
     },
     new Response.ErrorListener() {
       @Override
       public void onErrorResponse(VolleyError error) {
       error.printStackTrace();
       }
     })
```

Notice that the app uses `HTTP POST` to send data and it overrides two methods:

- `onResponse` that is invoked when the HTTP response is available
- `onErrorResponse` that is invoked if the request gets an error response

We can use these two methods to log the errors or somehow notify the error to the user.

To inform the user that the app is sending data to Artik, we can use an LED that turns on during the send phase. If an error occurs, the app could turn on another LED to notify the user that there are problems.

@Override
 public Map<String, String> getHeaders() throws AuthFailureError {
 Log.d(TAG, "Get headers..");
 Map<String, String> headers = new HashMap<String, String>();
 headers.put("Content-Type", "application/json");
 headers.put("Authorization", "Bearer " + token);
 return headers;
}

Moreover, in the preceding code, the app adds the request `Content-Type` setting it to application/JSON.

Send the data using a custom body request

The last step is implementing the custom request body. The request body represents the data the app sends, so it must have a structure compliant with the specification described previously (see the message structure described in the previous paragraphs). To do it, the app overrides the `getBody()` default method:

```
@Override
public byte[] getBody() throws AuthFailureError {
Log.d(TAG, "Creating body...");
try {
JSONObject jsonRequest = new JSONObject();
jsonRequest.put("sdid", deviceId);
jsonRequest.put("ts", System.currentTimeMillis());
JSONObject data = new JSONObject();

data.put("Temperature", temp);
data.put("Pressure", press);
jsonRequest.put("data", data);
String sData = jsonRequest.toString();
Log.d(TAG, "Body:" + sData);
return sData.getBytes();
}
catch (JSONException jsoe) {
jsoe.printStackTrace();
}
return "".getBytes();
}
```

The method uses the JSON library to build the JSON message. Notice it adds to the message all the parameters described previously.

The last thing to do is add the request to the queue so that `Volley` will manage it: queue.add(request);

That's all; the Artik client is now ready.

Sending data from the Android Things app

Once the client is ready, we have to call it from `MainActivity.java`, the class that we used to read sensor data. The easiest way to send data to Artik Cloud is invoking its API whenever the sensor reads a new value. Anyway, we have to consider the high frequency at which the sensor reads new values. This approach would require calling the Artik API almost continuously. The best approach is sending data using a scheduler. With a scheduler, the Android Things app sends data at specific time intervals without overwhelming the Artik Cloud. In this way, we can adjust the frequency having more control on the app behavior and the bandwidth the app consumes. Let us modify `MainActivity.java`:

1. Add the following method to this class:

```
// Scheduler to send data//
        private void initScheduler() {
          ScheduledExecutorService scheduler=
          Executors.newSingleThreadScheduledExecutor();

         scheduler.scheduleAtFixedRate(new Runnable() {
         @Override
         public void run() {
          double mTemp = totalTemp / tempCounter;
          double mPress = totalPress/
          pressCounter * FACTOR;
          totalTemp = 0;
          totalPress = 0;
          tempCounter = 0;
          pressCounter = 0;
         // call artik
         ArtikClient.getInstance(MainActivity.this,
         DEVICE_ID, TOKEN).sendData(mTemp, mPress);
          }
         }, 1, TIMEOUT, TimeUnit.MINUTES);
        }
```

In this class, we use a `SchedulerExecutorService` that runs a specific task continuously with a delay specified in `TIMEOUT`. The task is defined inside the `run()` method. In this method, the app executes the following steps:

2. Calculate the mean value of the temperature in the time interval between the last time the data was sent to Artik and the current time.
3. Calculate the mean value of the pressure in the same way as described in step 1.
4. Transform the pressure expressed in millibars into `mmHg` (the measuring unit specified in the Artik cloud console).
5. Call `ArtikClient` to send the mean value of the temperature and the mean value of the pressure.
6. The app resets the total value of the temperature and the pressure. At the same time, it resets the total counter of the temperature and the pressure samples acquired.
7. The last step is invoking this method in order to schedule the task. To this purpose, add the following:

```
        method into the onCreate() method:
         initScheduler();
```

You can configure the timeout parameter used by the scheduler. It represents the interval time between two task executions. Modifying the value of the timeout parameter, we can control how often the app sends

data to Artik Cloud. In this example, the app uses one minute of timeout. You can adjust it according to your need.

Now you can run the app and you will notice that it starts sending data. The following is the log written by the app:

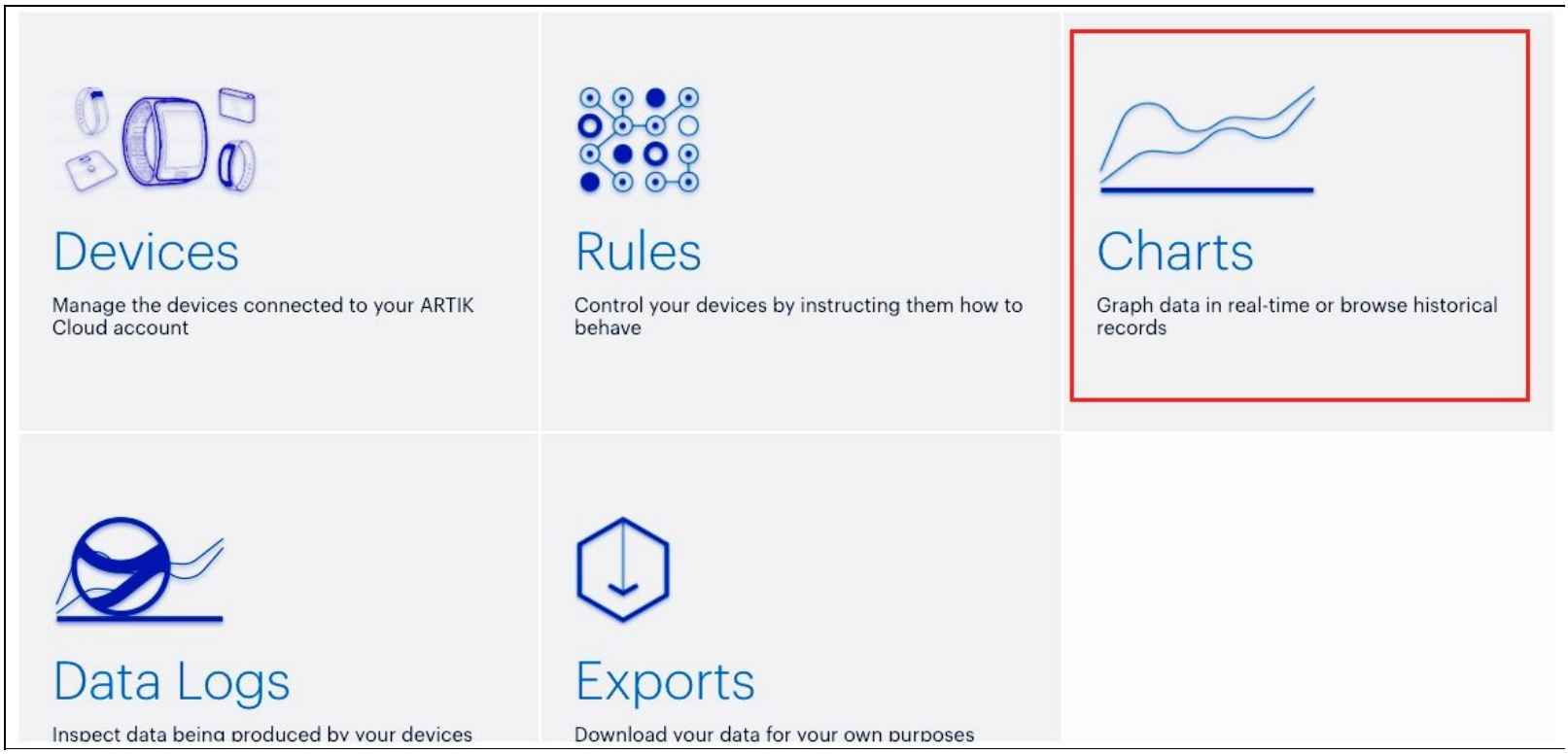
```
D/MainActivity: On Sensor connected... D/MainActivity: Temp sensor.. D/MainActivity:
On Sensor connected... D/MainActivity: Pressure sensor.. D/MainActivity: T. Accuracy [3] D/MainActivity: P. Accuracy
```

You can see the response coming from the Artik cloud platform informing the app that the data sent is acquired by Artik.

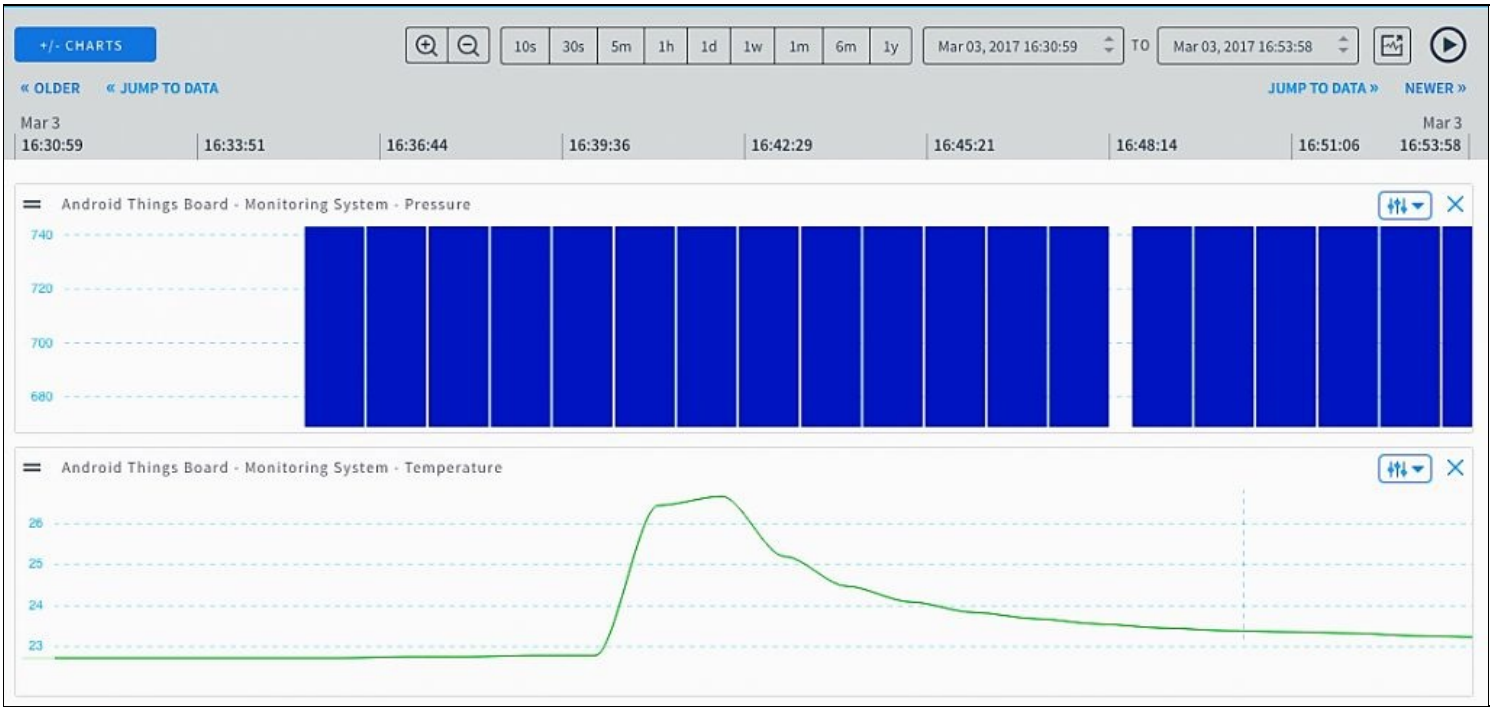
Creating a dashboard

While the app is running, it acquires data from sensors and sends the values acquired to the Artik cloud. We can use these values to create charts and visualize the data using different formats. Data charts offer a better way to analyze the data. Let us see how to do it:

1. Log in to the Artik cloud.
2. Go to https://my.artik.cloud/ and select Charts:



3. Click on Chart in the left top of the window to add the variable we used to collect data from the Android Things board. Add *temperature* and *pressure*.
4. Adjust the time range to fit the period when you sent the data.
5. At the end, you will see a chart like the following:



6. Notice the temperature, the pressure, and the different charts used to represent these two variables.
7. You can use other chart types to better represent your information.

Data logging

While charts give an overview of the values acquired by the Android Things app we developed, we can use other ways to show data. We can visualize all the requests Android Things clients made to the Artik Cloud. To do it, follow these steps:

1. Go back to the Artik cloud dashboard.
2. Select **DATA LOGS** and the platform will visualize all the events (or requests) received:

DEVICE	RECORDED AT	RECEIVED AT	DATA
Android Things board - Monitoring system	Mar 3 2017 16:44:04.523	Mar 3 2017 16:44:05.225	{"Pressure":743.2368145808717,"Temperature":24.4830662
Android Things board - Monitoring system	Mar 3 2017 16:43:04.523	Mar 3 2017 16:43:05.196	{"Pressure":743.2063349270118,"Temperature":25.1952328
Android Things board - Monitoring system	Mar 3 2017 16:42:04.523	Mar 3 2017 16:42:05.196	{"Pressure":743.1804380089573,"Temperature":26.6807213
Android Things board - Monitoring system	Mar 3 2017 16:41:04.523	Mar 3 2017 16:41:05.244	{"Pressure":743.2000930085092,"Temperature":26.4473325
Android Things board - Monitoring system	Mar 3 2017 16:40:04.524	Mar 3 2017 16:40:05.199	{"Pressure":743.2602728650169,"Temperature":22.7661898
Android Things board - Monitoring system	Mar 3 2017 16:39:04.527	Mar 3 2017 16:39:05.211	{"Pressure":743.2648839078688,"Temperature":22.7591543
Android Things board - Monitoring system	Mar 3 2017 16:38:04.526	Mar 3 2017 16:38:05.343	{"Pressure":743.2618311068334,"Temperature":22.7484212

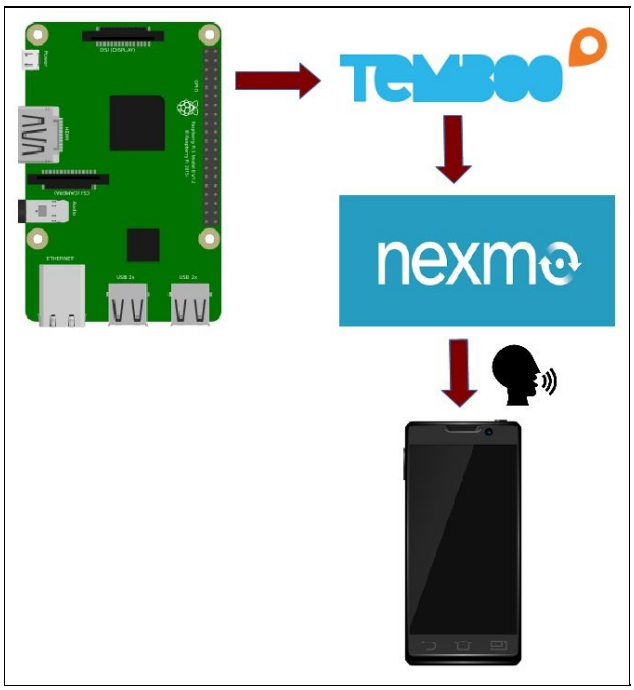
The column meanings are easy to understand. The first one is the device we have configured in the previous paragraphs while the last one is the data we sent. You can see the JSON that the Android Things app sent to the cloud.

There are other features you can explore such as exporting the data and so on. You can even modify the Android Things app to send, not only temperature and pressure, but also the humidity. In this scenario, you have to modify the *Manifest*, adding a new variable to hold the values.

Adding voice capabilities to Android Things

By now we have described how to send data to IoT cloud platforms. In this context, an IoT platform behaves like a data container where we store information. There are other kinds of services offered by IoT platforms. There are some IoT platforms that provide integration services. In other words, they are not focused on acquiring data from sensors and storing it, but their target is offering integration services with other cloud systems. One of these platforms is *Temboo* (https://temboo.com/). It offers a long list of integration services that can be used to extend the capabilities of an IoT app. Generally speaking, **Temboo** supports several programming languages and IoT platforms and one of these OSes is Android. This is perfect for our project.

What we want to do is add voice capabilities to our Android Things app so that it can trigger a voice phone call with a pre-configured message to inform us that a specific event happened. To this purpose, the Android Things app uses a Temboo service, called **choreo**, which simplifies the integration with Nexmo. **Nexmo** (https://www.nexmo.com/) is a voice cloud platform. The following schema describes the overall integration architecture that we will use to add this new feature to our Android Things app:



In the preceding figure, the Android Things app (represented by Raspberry PI) invokes Temboo when an event occurs. In this example, we invoke Temboo when the temperature is over 5°C. The app uses the Temboo platform just as an integration service. In turn, Temboo invokes Nexmo, which makes the phone call. Nexmo uses the TTS (Text To Speech) engine to convert the text to the human voice that we hear during the phone call.

The steps to follow to add these new features are:

1. Configure Temboo choreo to talk to Nexmo.
2. Modify the Android Things app to integrate Temboo choreo.

Let us see how to do it.

Configure Temboo choreo

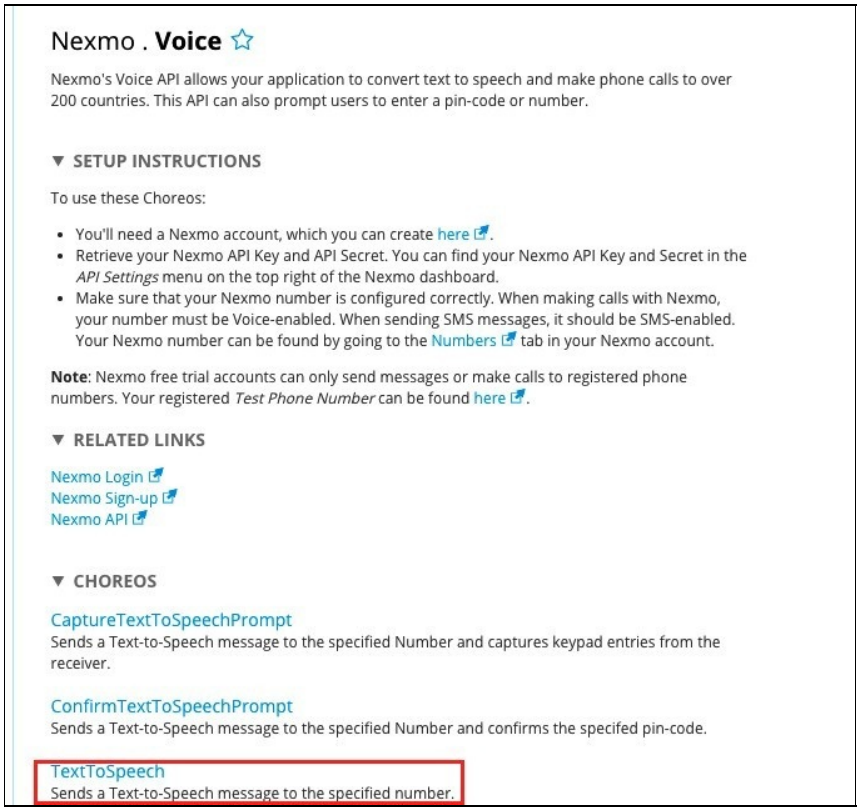
In this step, we have to configure the Temboo platform to invoke the Nexmo API service. The first step is creating a free Nexmo account that we use to test our app. Once you have created an account, you will have access to the console dashboard. We use it just to get two pieces of information:

- Key
- Secret

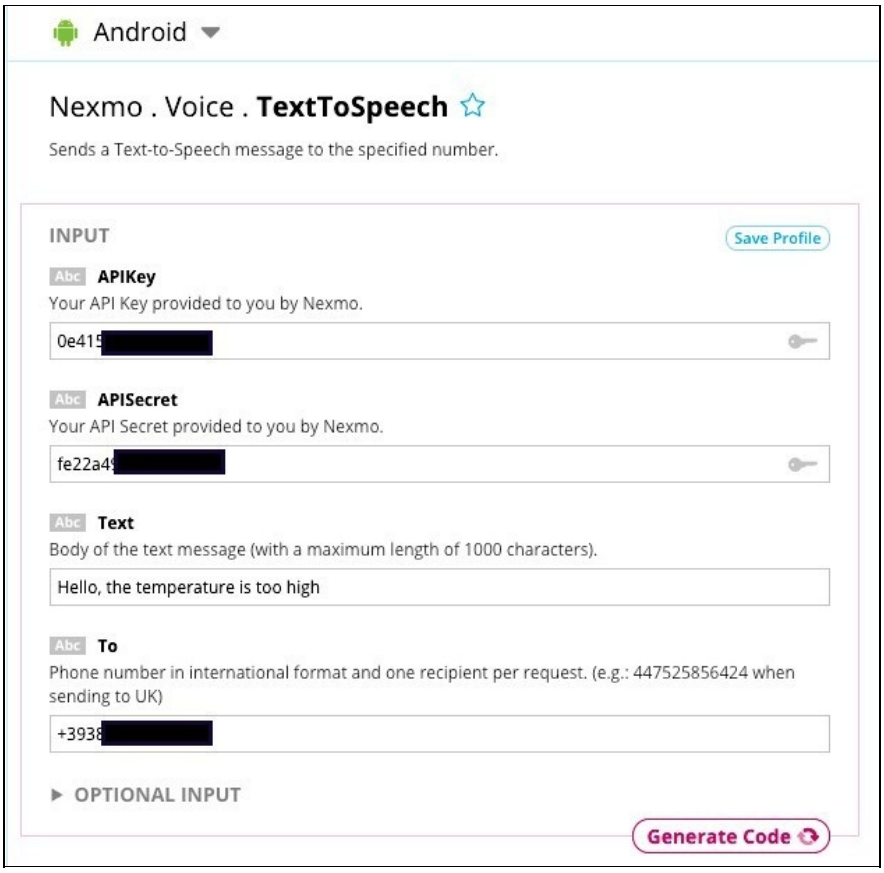
The following screenshot shows the Nexmo dashboard with the two parameters:

They are used by Temboo to authenticate the service request. Now we have to create a Temboo account if you do not have one already. Once you are logged in, follow these steps:

1. Look for Nexmo choreo and select it:



2. Once you have selected TextToSpeech, you will get a page like the one shown in the following screenshot:



3. Now, you have to provide:
    1. The key and secret you got previously from Nexmo.
    2. The text that will be read by Nexmo.
    3. The phone number.

    It is important that you select Android on the top.

Now you can click on **Generate Code** to get the snippet to use in our Android Things app:

That's all the configuration steps completed. Now we can integrate it in our Android Things app.

Integrate Temboo in the Android Things app

In this last step, we modify the Android Things app to include the snippet we got previously, and handle the integration with Temboo:

1. Open Android Studio to modify the app.
2. Download from this link (**https://temboo.com/download**) the Temboo SDK for Android:

3. Add the Temboo libraries to project libraries. You have to add:
   - `temboo-android-sdk-core-xxx.jar`
   - `Nexmo-xxx.jar`
4. Add a new class called `TembooClient.java` to the project that will handle the Temboo integration details. The core of this class is the method shown here that invokes the Temboo choreo:

```
public void callTemboo() {
          Runnable r = new Runnable() {
          @Override
          public void run() {
            Log.d(TAG, "Call Temboo...");
            TextToSpeech textToSpeechChoreo = new
                    TextToSpeech(session);
            TextToSpeech.TextToSpeechInputSet
                    textToSpeechInputs =
                    textToSpeechChoreo.newInputSet();
            textToSpeechInputs.set_APIKey("xxxx");
                    textToSpeechInputs.set_Text("Hello,
            the temperature is too high");
            textToSpeechInputs.set_To("xxxx");
            textToSpeechInputs.
                set_APISecret("1xxx");
            try {
               TextToSpeech.TextToSpeechResultSet
                 textToSpeechResults =
                 textToSpeechChoreo.
                   execute(textToSpeechInputs);
                 Log.d(TAG, "TTS Result
          ["+textToSpeechResults.get_Response()+"]");
                 }
                 catch (TembooException te) {
                  te.printStackTrace();
                }
              }
            };
         Thread t = new Thread(r);
                t.start();
         }
```

This method wraps the choreo code that we configured in the previous steps.

5. Finally, we invoke this class in the `onSensorChanged` in `TemperatureCallback`:

```
if (val >= 5) {
            TembooClient client = TembooClient.getInstance();
            client.callTemboo();
        }
```

That's all. Now we can run the app and verify that, when the temperature is over 5°C, the app invokes Temboo and we get a phone call. We can improve the code shown previously with a control that avoids sending repeat phone calls every time the temperature is over the threshold in the last pre-configured time interval.

You can extend this project integrating other services.

Summary

In this chapter, we covered how to integrate Android Things with IoT cloud platforms. Moreover, we have covered how to stream data to the cloud and how to create dashboards using the data sent. Now we are ready to use other kinds of IoT platforms and integrate the Android Things boards with cloud services. The concepts developed in this chapter can be applied to other IoT platforms even if they have different features and capabilities. In the next chapter, we will learn how to use Android Things to control remote IoT boards that do not support Android Things. We will implement a master-slave architecture where the Android Things board is the master and the slaves are the low-level boards such as Arduino UNO.

Create a Smart System to Control Ambient Light

This chapter describes how to create a system that uses **Android Things** to control ambient light. During this project, we will explore how to use the Android Things app to control **IoT** boards such as Arduino. As we will see later, Android Things can be used as anÂ *IoT gateway* that manages one or more remote IoT boards in a *master-slave* architecture.

The main topics covered in this chapter are:

- How to use Android Things in a master-slave architecture
- Implementing an Arduino sketch to control a RGB LED strip
- How to use HTTP protocol to exchange data
- Creating an Android Things UI

At the end of this chapter, we will build a real-life working IoT project based on Android Things that we can use in our home to manage LED lights.

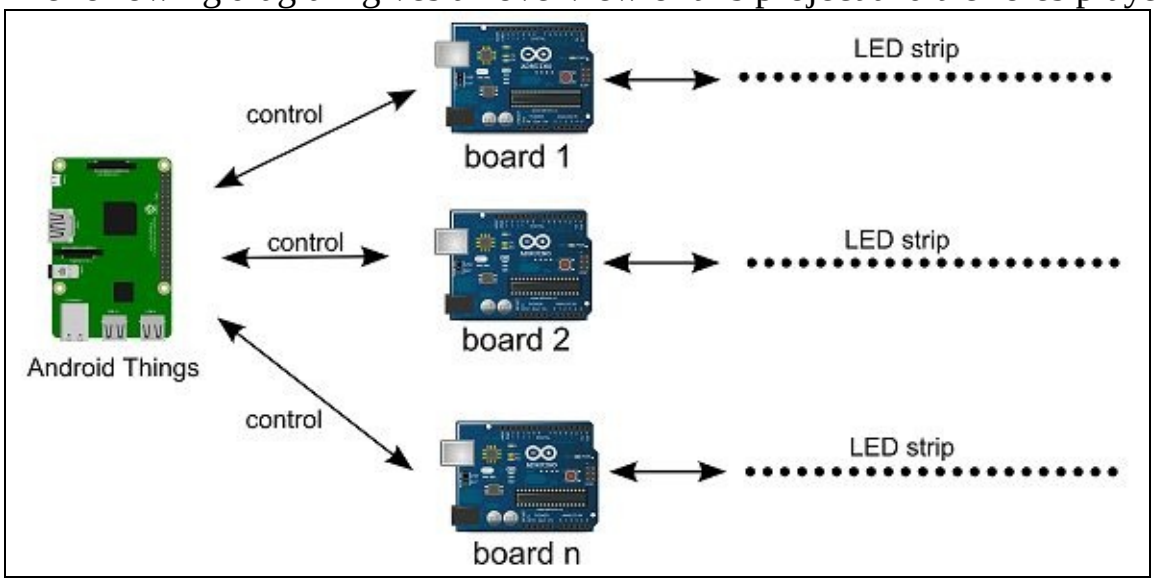
Ambient light control system description

Before diving into the project implementation details, it is useful to have an overview of the project we want to build. The idea that stands behind this project is building a system that has a unique center of control, represented by Android Things, and several remote IoT boards that are connected to RGB LED strips. These IoT boards receive the commands from the Android Things app and according to these commands, they set various RGB LED color or apply several light effects.

This project uses two different IoT boards:

- Android Things compatible boards such as Raspberry
- PI 3 Arduino Uno R3

The following diagram gives an overview of this project and the roles played by these two boards:



As you can notice, the LED strips are connected to Arduino boards that manage them directly. In turn, the Arduino boards receive commands from the Android Things board using HTTP protocol. HTTP protocol is a web-oriented protocol that we can use to exchange data between different IoT boards. Even if it introduces some overheads and it is a general purpose protocol, it is easy to implement and we can easily handle it in Android Things and Arduino. For some contexts, the HTTP protocol would not be the best choice, especially when there is the need to publish data in a one-to-many paradigm. In this project, anyway, we do not need these features. In the next chapter, we will cover a more specific IoT protocol called MQTT.

Generally speaking, when we build a project where we have to implement a Request-Response paradigm, where there are not network constraints and we want to use a well-known and widespread protocol, HTTP protocol can be our ally.

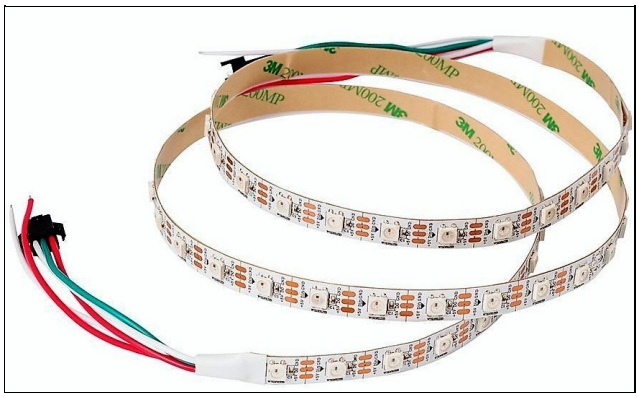
The benefits of this architecture are:

- Unique central point of control
- Unique interface to external world
- Easy to implement

Project components

To build this IoT project, we need:

1. LED Strip. In this project, we will use an individually addressable LED strip based on *WS2812b* protocol. The following is an image of one type of this LED strip:



Source: https://www.amazon.it/gp/product/B01CDTE9UC/ref=oh_aui_detailpage_o00_s00?ie=UTF8&psc=1

2. This is a 60 RGB LED strip and its length is 1 meter. You can buy other types of LED strips, but they must use WS2812b protocol because this protocol guarantees we can address a single LED in the strip. According to the manufacturer specifications, this strips requires 18W, so we need an external power supply.

3. To power on the RGB LED strip a power supply is necessary. The LED strip requires 18W at 5V therefore we need a power supply with these specifications. The following image shows the one used in this project:

Source https://www.amazon.it/gp/product/B01HRR9GY4/ref=oh_aui_detailpage_o00_s00?ie=UTF8&psc=1

4. An Arduino board (one or more). This project uses Arduino Uno R3, but you can use other board types compatible with Arduino. If you use Arduino Uno you need a WiFi shield or an Ethernet shield to connect this board to the net.

Project architecture

Now we know the components that will build our IoT project we have to understand how these components are connected. In this project, the Android Things board does not manage sensors or other kinds of peripherals; it acts as a central gateway that handles remote boards. Therefore, the project can be divided into two different components:

1. Arduino project.
2. Android Things app.

It is important to clarify the roles played by these two blocks in order to have a clear overview when we develop the project.

The main tasks that the Arduino board has to do are:

1. Implement the logic to handle the RGB LED strip according to the WS2812 protocol.
2. Expose a set of services that can be called and used by the Android Things app.

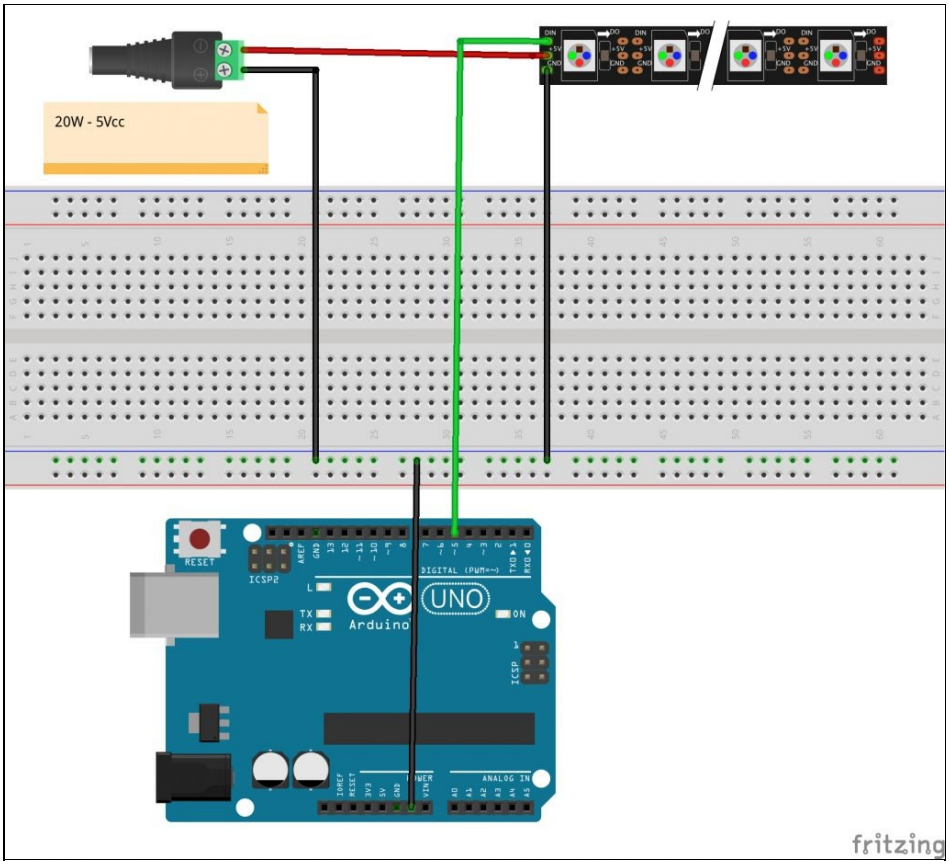
On the other hand, the tasks of the Android Things app are:

1. Implement the logic to invoke the services exposed by the Arduino board.
2. Implement a User Interface so that a user can remotely control the RGB LED strip.

Let us start by implementing the Arduino project side.

Building the Arduino project

In this paragraph, we will implement the Arduino project. The first step is connecting the RGB LED strip to Arduino, as described in the following figure:



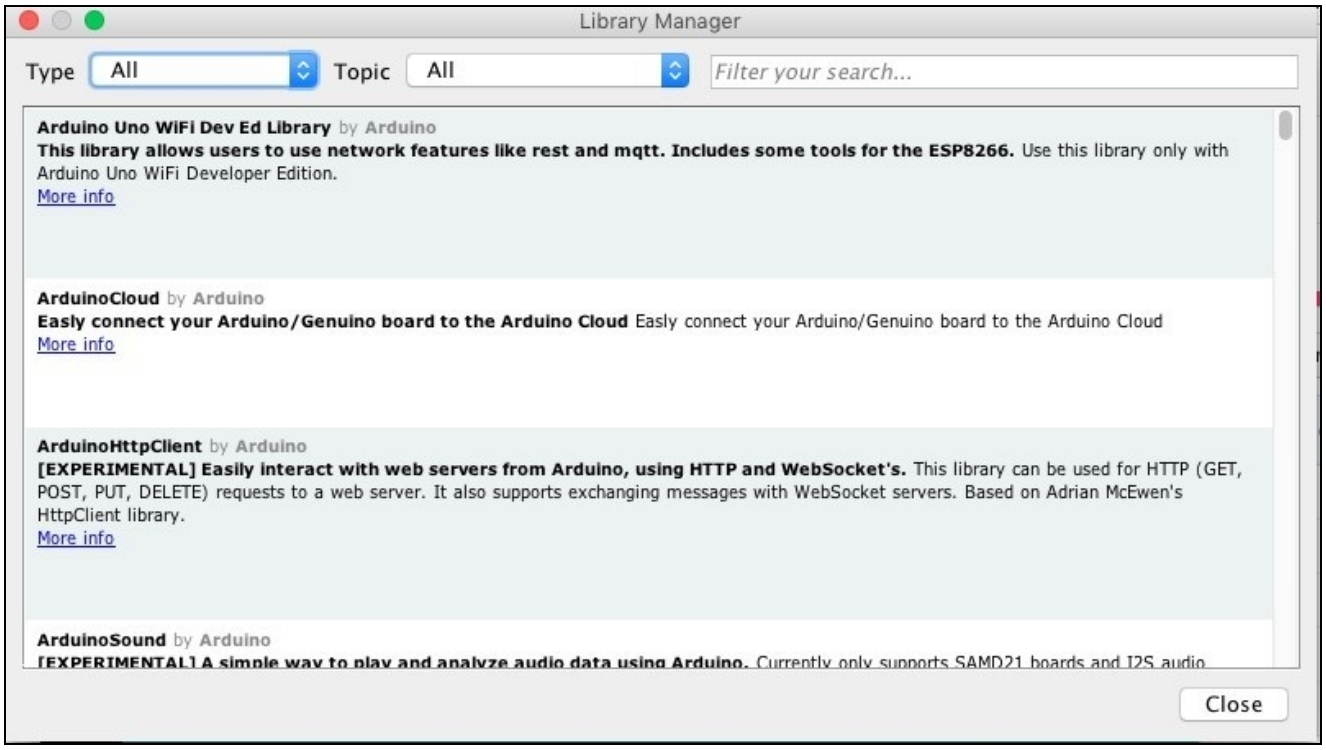
As you can see, the wiring is very simple. The WS2812 peripherals have only one data pin that is connected to the Arduino pin **5** (PWM).

Do not forget to connect the ground all together to have a common reference.

Now it is time to start developing the code. To handle the RGB LED strip we will use a library developed by **Adafruit** that makes it very easy to manage every RGB LED available on the strip. You can directly use the source code of this sketch and upload it into your Arduino Uno board, skipping all the following steps. You can find the source code once you have downloaded the book project source code. Anyway, it is useful to follow this guide step by step because it helps you to better understand how to implement the Android Things app.

Therefore, the first step is installing this library:

1. Open Arduino IDE.
2. Navigate to Sketch | Include Library | Manage Library and at the end, you will have a new window like the one shown in the following screenshot:



3. In the input box write `neopixel`. The IDE will show a list of libraries. Select the library called `Adafruit Neopixel`.
4. Click on the Install button.

The library is ready and we can use it in our sketch. At the beginning, we will use the library to manage the RGB LED color and to create simple effects. In the next steps, you have to use the Arduino IDE. If you do not have it already you can download this IDE from https://www.arduino.cc/en/main/software. The steps are described as follows:

1. Create a new sketch in Arduino IDE.
2. Add at the top of the file the following line:

```
#include <Adafruit_NeoPixel.h>
```

3. Now we have to define the pin where the LED strip is connected and the number of LEDs available. Therefore add the following line:

```
#define PIN 5
        #define LED_NUMBER 60
```

In the strip we are using in this project, there are 60 RGB LEDs; change this value according to the number of LEDs available in your strip.

1. Now the sketch configures the communication with the LED strip:

```
Adafruit_NeoPixel strip =
        Adafruit_NeoPixel(LED_NUMBER, PIN,
        NEO_GRB +NEO_KHZ800);
```

You can find more information about how to set the configuration parameters at this link https://github.com/adafruit/Adafruit_NeoPixel.

2. In the `setup()` method add the following line:

```
strip.begin();
```

   This line initializes the strip so that we can start using it.

3. Now add the following method to the sketch:

```
void fillStrip(uint32_t color, int wait, int
       direction) {
         int first, last;
         setDirection(&first, &last, direction);
         for (int p = first; p <= last; p++) {
            strip.setPixelColor(abs(p), color);
            strip.show();
           delay(wait);
          }
        }
```

   This method simply fills the strip with a color passed in the `uint32_t color` parameter. Inside this method, we apply a simple animation when we turn on every RGB LED in the strip. The `wait` parameter represents the time elapsed before the next RGB LED is turned on by modifying this parameter we can increase or decrease the time spent to fill the strip with the specified color. In addition to this, the `direction` represents the way the RGB LEDs are turned on: from the bottom to the top or vice-versa.

4. Finally, the last method turns all the RGB LEDs off using the same effects described previously:

```
void clearStrip(int wait, int direction) {
          int first, last;
          setDirection(&first, &last, direction);
            for (int p = first; p <= last; p++) {
            strip.setPixelColor(abs(p), 0); strip.show();
            delay(wait);
           }
        }
```

You can add other methods to implement different light effects. In the source code, it also implemented a rainbow effect. Before proceeding further, it is worthwhile to test the preceding code to be sure that Arduino manages the RGB LED strip correctly.

#include <Ethernet.h>
 #include "aREST.h"

#define SERVER_PORT 80

EthernetServer server(SERVER_PORT);
 // Create aREST
 aREST rest = aREST();

rest.function("fill", setStripColor);
 rest.function("clear", setClearStrip);
 rest.function("rainbow", setRainbow);

int setStripColor(String command) {
 Serial.println("Color strip function...");
 struct data value = parseCommand(command);
 debugData(value);
 fillStrip(strip.Color(value.r,value.g,value.b),
 value.wait, value.dir);
 return 1;
 }

In the previous code, the `parseCommand` function simply extracts the data we will use to manage the strip. In this project, we suppose that the command structure is very simple and it is represented by a string where all the values are chained. The structure is:

- The first char represents the direction
- The next 6 chars represent the color in hex format
- The next 2 chars the delay
- The last one the function

At the end, this function calls the `fillStrip` function to set the strip color effects. In the same ways are defined the other methods exposed. You can find the code in the companion project example attached to this book.

Now we are ready to develop the Android Things app to control the Arduino board.

Implementing the Android Things app

Let us come back to the Android Things app. Once the Arduino sketch is implemented and works correctly, we can focus our attention on the Android Things side. The app we are going to develop has to control the Arduino board and in turn the RGB LED strip. To this purpose the app must have:

- A user interface to interact with the user, so that they can select the strip LED color or activate an effect
- Exchange data with remote boards using the services exposed as described in the previous section

An important aspect is related to the user interface. As stated in Chapter 1, *Getting Started with Android Things*, the user interface is optional. This means that there are some devices that support the UI and other devices that do not support it. For example, Raspberry PI 3 belongs to the group that supports the UI while Intel Edison with Arduino breakout kit will not. For this reason, we will run the app on the Raspberry PI 3, while for Intel Edison we will use a different approach that we will describe later.

<RelativeLayout xmlns:android
 ="http://schemas.android.com/apk/res/android"

 android:layout_width="match_parent"
android:layout_height="match_parent">

<TextView
 android:layout_width="wrap_content"
android:layout_height="wrap_content"
 android:text="R"
android:layout_below="@id/txtLabel"
 android:layout_margin="10dp"
android:id="@+id/lblRed"/>
 <SeekBar
 android:layout_width="200dp"
android:layout_height="wrap_content"
 android:max="255"
android:id="@+id/rColorBar"
 android:layout_alignBottom="@id/lblRed"
android:layout_toRightOf="@id/lblRed"/>
 <TextView
android:layout_width="wrap_content"
 android:layout_height="wrap_content"
android:text="G"
 android:layout_below="@id/lblRed"
android:layout_margin="10dp"
 android:id="@+id/lblGreen"/>
<SeekBar
 android:layout_width="200dp"
android:layout_height="wrap_content"
 android:max="255"
android:id="@+id/gColorBar"
 android:layout_alignBottom="@id/lblGreen"
android:layout_toRightOf="@id/lblGreen"/>
 <TextView
android:layout_width="wrap_content"
 android:layout_height="wrap_content"
android:text="B"
 android:layout_below="@id/lblGreen"
android:layout_margin="10dp"
 android:id="@+id/lblBlue"/>
 <SeekBar
android:layout_width="200dp"
 android:layout_height="wrap_content"
android:max="255" android:id="@+id/bColorBar"
android:layout_alignBottom="@id/lblBlue"
android:layout_toRightOf="@id/lblBlue"/>

<TextView
 android:layout_width="wrap_content"
android:layout_height="wrap_content"
 android:text="Delay in milliseconds"
android:id="@+id/lblDel"
 android:layout_below="@id/lblBlue"
android:layout_marginTop="20dp"/>
 <EditText
android:layout_width="wrap_content"
 android:layout_height="wrap_content"
android:layout_below="@id/lblDel"
 android:text="10"
android:id="@+id/delText"/>

<TextView
 android:layout_width="wrap_content"
android:layout_height="wrap_content"
 android:text="Direction"
android:id="@+id/lblDir"
 android:layout_below="@id/delText"
android:layout_marginTop="20dp"/>
 <Spinner
android:layout_width="wrap_content"
 android:layout_height="wrap_content"

android:id="@+id/direction"
 android:layout_below="@id/lblDir"/>

<Button
 android:layout_width="wrap_content"
android:layout_height="wrap_content"
 android:layout_below="@id/txtLabel"
android:layout_marginTop="25dp"
 android:layout_marginRight="25dp"
android:text="Go!"
 android:id="@+id/btnGo"
android:layout_alignParentRight="true"/>
 <Button
android:layout_width="wrap_content"
 android:layout_height="wrap_content"
android:id="@+id/btnClear"
 android:text="Clear the strip"
android:layout_below="@id/btnGo"
 android:layout_alignLeft="@id/btnGo"/>
<Button
 android:layout_width="wrap_content"
android:layout_height="wrap_content"
 android:id="@+id/btnRainbow"
android:text="Rainbow"
 android:layout_below="@id/btnClear"
android:layout_alignLeft="@id/btnGo"/>

That's all; the layout is defined.

Attaching the layout to the Activity

To show the layout when the Activity is started we have to attach the layout defined previously to the Activity itself:

1. Open `MainActivity.java` and in the `onCreate` method add the following line:

```
setContentView(R.layout.main_activity);
```

   In this way, the layout defined previously is attached to the Activity.

2. Now we have to get a reference to each widget. We do it using `findViewById` that accepts the `id` used in the layout to identify the widget. For example, to get the reference to the red seek bar we use:

```
rBar = (SeekBar) findViewById(R.id.rColorBar);
```

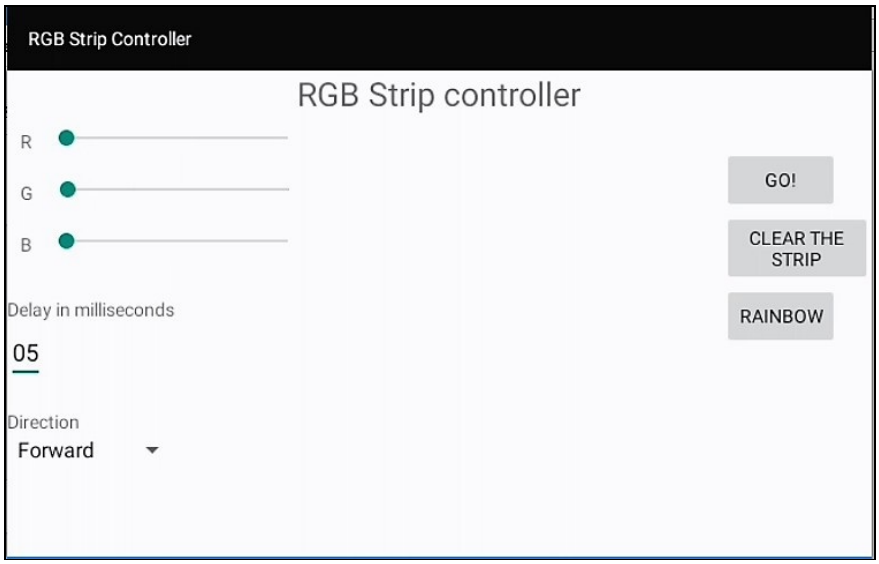
3. Let us repeat the preceding code for all the widgets in the layout. Be aware that the widget type is not always the same.
4. In the layout described before, we have used a `Spinner` widget. This type of widget must be populated with the values so that the user can select one of them. To do it, we use an *Adapter*. This is a bridge between the `View` (or the widget) and the underlying data. You can find more information at this link (https://developer.android.com/reference/android/widget/Adapter.html). In this example, we use a simple `ArrayAdapter`:

```
        dirSpinner = (Spinner)
        findViewById(R.id.direction);
        ArrayAdapter<CharSequence> adapter =
        ArrayAdapter.createFromResource(this,
        R.array.direction,
        android.R.layout.simple_spinner_item);
        adapter.setDropDownViewResource(android.R.layout.simple_spinner_dro pdown_item);
        dirSpinner.setAdapter(adapter);
```

   Here, `R.array.direction` is defined in the `String.xml` file under the values folder in this way:

```
<string-array name="direction">
        <item>Forward</item>
        <item>Backward</item>
        </string-array>
```

Now the layout definition is completed and the layout is attached to the Activity. If we run the Android Things app on Raspberry PI 3, we get something like the following:



It is time to handle the UI events.

Handling UI events

In order to handle the user interaction with the widget, it is necessary to attach a listener to the widget. There are several kinds of listeners and we have to use the right one according to the type of event we want to listen to.

Let us start with the seek bar. In this case, we want to be informed when the progress level in the bar is changing:

1. Look for the `rBar` attribute defined in the `MainActivity`.
2. Attach this listener to `rBar`:

```
rBar.setOnSeekBarChangeListener(new
        SeekBar.OnSeekBarChangeListener() {
        @Override
        public void onProgressChanged(
        SeekBar seekBar, int i, boolean b) { red = i;
        }
        @Override
        public void onStartTrackingTouch(SeekBar seekBar)
       { }
        @Override
        public void onStopTrackingTouch(SeekBar seekBar)
        {}
        });
```

   As you can see, we are interested only in the `onProgressChanged` method, which is called when the bar value changes. In this case, considering we are handling the red seek bar, we simply save the current bar value to a global variable called `red`.

3. Repeat the same piece of code for the other two seek bars, storing the green and blue values in two global variables.

   Now, we have to focus our attention on buttons. We use them to send commands to the Arduino board and in turn to the RGB LED strip. In this context, we are interested to be informed when the user clicks on the button so that we have to use the right listener:

4. Get the reference to the button widgets, as described previously:

```
btnGo = (Button) findViewById(R.id.btnGo);
        btnClear = (Button) findViewById(R.id.btnClear);
        btnRainbow = (Button)
        findViewById(R.id.btnRainbow);
```

5. For each button, we have to attach the listener. For `btnGo`, we add the following listener:

```
btnGo.setOnClickListener(new
        View.OnClickListener() {
        @Override
        public void onClick(View view) {
        // Call the Arduino board services
        }
        });
```

6. Let us repeat the same piece of code for the other two buttons.

Great! We are ready now to handle the connection to the Arduino board and invoke the services exposed previously.

Invoking the Arduino services

In this paragraph, we will explore how to invoke the Arduino services exposed using the Rest paradigm. In this context, we have to call the services passing the data defined in the user interface exposed by our Android Things app. While in the previous chapter we used the Volley library, in this chapter we will use another library to handle the HTTP connection called `OkHTTP` (http://square.github.io/okhttp/). In this way, you use another approach and you can select the best one according to your requirements.

The first thing to do is add the library to our `build.gradle` file:

```
compile 'com.squareup.okhttp3:okhttp:3.6.0'
```

Now we can create another class to handle the communication details with the Arduino board:

1. Create a new class called `BoardController.java`.
2. Add a private constructor because this class must be a *singleton:*

```
private BoardController() { client = new
      OkHttpClient();
        }
        public static BoardController getInstance() { if
     (me == null)
        me = new BoardController(); return me;
        }
```

Notice that in the constructor we initialize the library that handles the HTTP connections.

3. Now we have to create a method that sends data to the Arduino board invoking the `Rest` services. The code may seem complex, but it is very simple and follows these steps:
    1. Convert the R,G,B values into the hexadecimal representation.
    2. Select which service to call. In this example, there are three different services: Set the strip color, clear the strip, and the rainbow effect.
    3. Make the request:

```
        public void sendData
                    (int r, int g, int b, int wait, int dir,
                 int func) { String
                   hexColor =getHex(r) + getHex(g) + getHex(b);
                       String params = Integer.toString(dir) +
                        hexColor + (
                       (wait < 10) ? "0" +
                 Integer.toString(wait) :
                       Integer.toString(wait)) + "9";
                    String url = baseUrl;
                switch (func) {case 0:
                        url += "fill";
                break; case 1:
                         url += "clear";
                break; case 2:
                         url += "rainbow";}
                Log.d(TAG, "URL ["+url+"] - Params
               ["+params+"]");
                Request request = new Request.Builder()
                     .url(url + "?params=" + params)
                     .build();
                client.newCall(request).enqueue(new Callback() {
```

```
            @Override
                public void onFailure(
                Call call, IOException e) {}
                @Override
                public void onResponse(
                Call call, Response response) throws
              IOException {
                 Log.d(TAG, "Response
             ["+response.body().string()+"]");
                 }
                 });
```

Notice that the method creates the URL to call appending to the base URL, in other word, the IP address of the Arduino board, the param holding the value to send. Moreover, notice how simple it is to make the HTTP request. We simply invoke the URL and wait for the response defining a callback class.

4. Now, we have to retrieve the parameters defined in the app UI and send them to the Arduino services.
5. Now, let us modify the listeners attached to the buttons so that when they are clicked invokes the method to send data.

Congratulations! Now the project is complete and we can use it to control Arduino boards. You can extend this project to handle other types of boards. As long as the services exposed remain the same you can reuse the Android Things app to handle other IoT boards.

The following is the app log when it makes the request to the Arduino services and gets the response:

```
      URL [http://192.168.1.6/fill] - Params
    [1803780059]         Response [{"return_value": 1, "id": "", "name":
    "", "hardware":
      "arduino", "connected": true}
```

In the preceding example, the app invokes the fill services to set the RGB LED color.

How to implement a web interface

There are some devices that do not support the UI or times when we do not want to create an UI and we prefer to expose a web interface instead. This is possible in Android Things by implementing a simple HTTP Web server. In this paragraph, we will describe how to use a web interface to control the RGB LED string in the same way we did previously. The basic idea that stands behind this is creating an HTML page where the user can set the values to control the LED strip. To do it we have to follow these steps:

1. Create a HTTP server to handle the incoming requests.
2. Create an HTML page containing all the controls to configure the RGB LED strip.
3. Embed the HTTP server into the Android Things app.

Let us describe how to do it.

public class AndroidWebServer extends NanoHTTPD {
 .. }

public AndroidWebServer(int port, Context ctx) {
 super(port);
 this.ctx = ctx;
 try {
 start();
 }
 catch(IOException ioe) {
 Log.e(TAG, "Unable to start the server");
 ioe.printStackTrace();
 }

@Override
 public Response serve(IHTTPSession session) {
 Map<String, String> parms = session.getParms();
 String param = parms.get("params");
 String action = parms.get("action");
 String delay = parms.get("delay");
 String r = parms.get("red");
 String g = parms.get("green");
 String b = parms.get("blue");
 String dir = parms.get("dir");
 String content = null;
 if (action == null) {
 content = readFile().toString();
 }
 else {
 Log.d(TAG, "Action ["+action+"]");
 listener.handleCommand(Integer.parseInt(r),
 Integer.parseInt(g),
 Integer.parseInt(b),
 Integer.parseInt(delay),
 Integer.parseInt(dir),
 Integer.parseInt(action));
 }
 return newFixedLengthResponse(content);
 }

There are two aspects to notice:

1. If the action is null then the class reads the file that holds the HTML content.
2. Otherwise, it invokes a listener passing the parameters extracted from the request.

The web server implementation is ready and we have to embed it into our Android Things app.

Creating the HTML page with the UI

In this step, we will create the HTML page that holds all the controls to configure the RGB LED strip as we did previously when implementing the Android UI:

1. In Android Studio create a folder called assets under the app folder.
2. Inside the assets folder, copy the file `home.html` from the companion book source code.

As you may know already, the assets folder is used to store arbitrary files such as text files, audio files, and so on. The app can reference it using `AssetManager` at https://developer.android.com/reference/android/content/res/AssetManager.html.

So it is time to implement the `readFile()` method used by the HTTP server to serve the HTML page:

1. Open the AndroidWebServer.java class again.
2. Add the following method:

```
private StringBuffer readFile() {
            BufferedReader reader = null;
            StringBuffer buffer = new StringBuffer();
            try {
             reader = new BufferedReader(
               new InputStreamReader
                (ctx.getAssets().open("home.html"), "UTF-
                  8"));
             String mLine;
             while ((mLine = reader.readLine()) != null) {
             buffer.append(mLine);
             buffer.append("\n");
               }
             }
             catch(IOException ioe) {
             ioe.printStackTrace();
             }
             finally {
             // close the reader
             }
           return buffer;
         }
```

As you can notice in the previous code, the method reads the `home.html` storing it in a `StrinBuffer` and this buffer is ready to be served as a response from the browser request.

The last step is embedding the HTTP server into the Android Things app.

Embedding the HTTP Server into the Android Things app

Finally, the last step: the easiest one. We have to start and stop the web server and implement the listener, so that we get notified when the user sends data using the HTML page:

1. Open `MainActivity.java` again and modify it in this way:

```
public class MainActivity extends Activity
        implements
        AndroidWebServer.WebserverListener { ... }
```

Here, the `WeserverListener` is the callback interface.

2. In the `onCreate` method let's add these lines:

```
if (Boards.enableWebserver()) {
        aws = new AndroidWebServer(8180, this);
        aws.setListener(this);
        }
```

The preceding code simply checks the type of the Android Things board as we described in Chapter 2, *Creating an Alarm System Using Android Things*. If the board is one that does not support the UI interface, the app starts the web server:

Finally, the app implements the `callback` method called by the `AndroidWebServer` class when it is time to control the RGB LED strip:

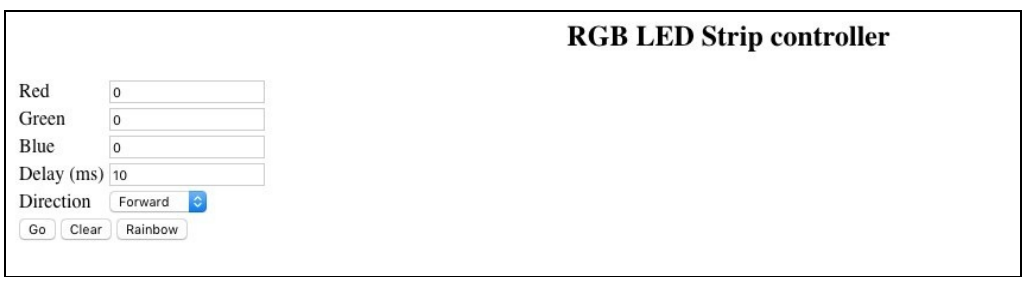
```
@Override
        public void handleCommand(int r, int g, int b, int
          delay, int
        dir,
        int func) {
           BoardController.getInstance().sendData(r, g, b,
                          delay, dir, func);
       }
```

In this method we simply send data to the `BoardController` that in turns invokes the Arduino services.

To test the app, we can open the browser and invoke the web server URL:

```
http://android_things_ip:8180/home.html
```

At the end, the Android Things app creates a web page that we can use to configure the RGB LED strip. The following screenshot shows the web interface:



Using the controls shown in the web page we can obtain the same result as if we were using the Android UI interface.

Summary

In this chapter, we explored how to use Android Things as a gateway to control IoT boards that are not compatible with Android Things OS. We discovered how to set up a master/slave architecture where Android Things acts as a master and as a front-end board. In the next chapter, we will explore another interesting aspect about how to use MQTT protocol in IoT projects and how to use Android Things and MQTT.

Remote Weather Station

This chapter explores how to build a Remote Weather station that acquires weather information using several sensors. This IoT project uses an Android Things-compatible board and several IoT boards that connect to Android Things using the MQTT protocol. Through this project, we will cover how to exchange data between different devices. This aspect is known as machine to machine communication. This is an important topic in the IoT ecosystem. As we will see during this chapter, **Machine to Machine** (**M2M**) includes all the technologies that enable devices to talk to each other. In this chapter, we will focus on the MQTT protocol, describing how it is used in real-life IoT projects.

In more detail, this chapter focuses on:

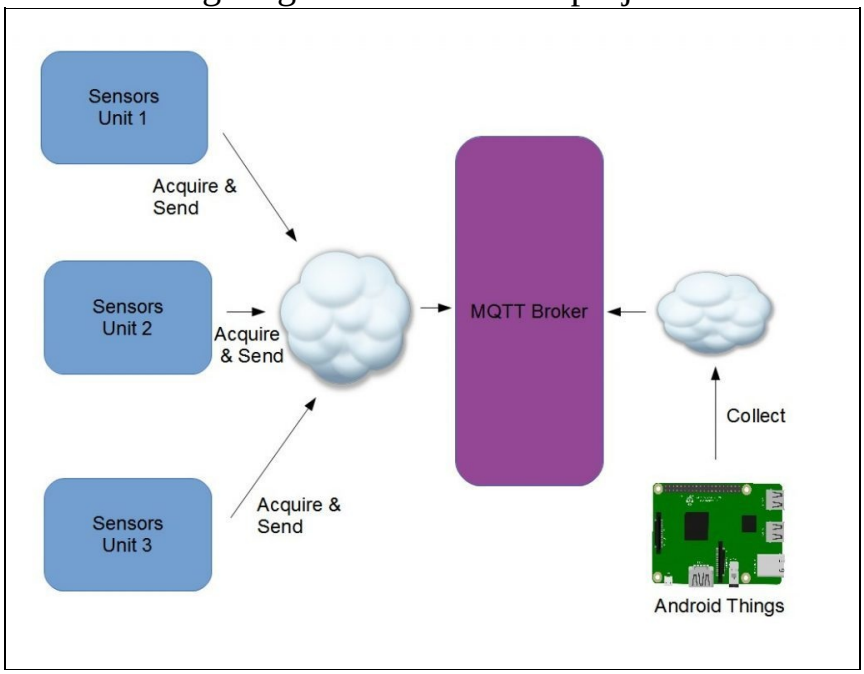
- The M2M architecture and MQTT protocol
- How to use the MQTT protocol with Android Things
- How to acquire and stream real-time data

Before starting, it is useful to have an overview of the Weather remote IoT project so that we can better understand theÂ *M2M architecture* and the role of the MQTT protocol.

Remote weather station project description

As stated before, we want to build an Android Things Remote weather station that acquires data from remote sensors connected to several IoT boards. In this project, the Android Things board acts as an MQTT client that collects data coming from remote sensors and visualizes it through an UI. In this context, the sensors are not connected physically to the Android Things board, but they are managed by other non-compatible IoT boards. In turn, these IoT boards exchange data with Android Things using MQTT. We will describe in more details what MQTT is and how to use it later. For now, it is enough to know that MQTT is a lightweight protocol that is widely used in M2M communication.

The following diagram describes the project architecture:



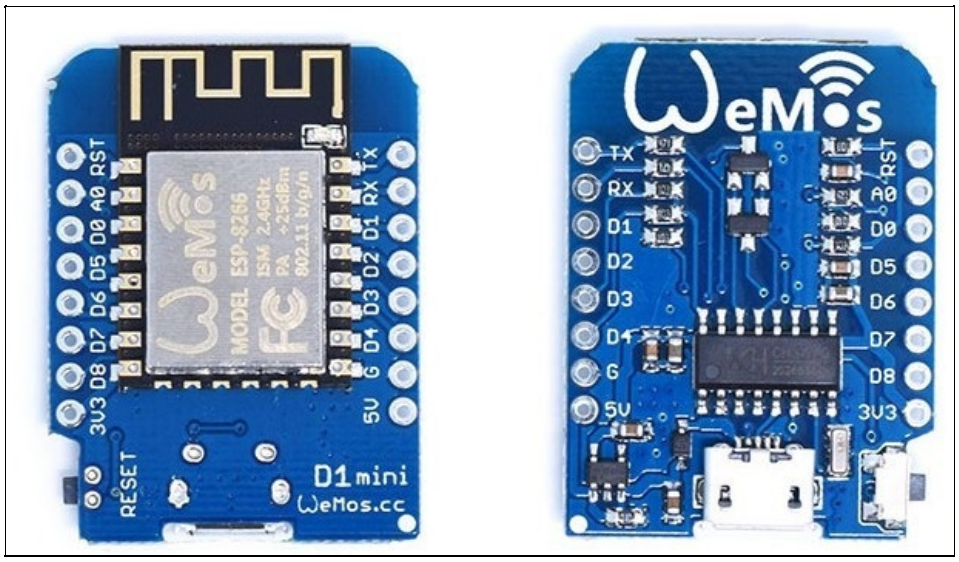
As you might already foresee, this project emulates the scenario where we have sensors and IoT boards physically placed somewhere far from the place where we collect data and analyze it. Therefore, this project architecture can be extended to other kinds of scenario as we will see later. This is a *Machine to Machine* architecture where data walks from the source (sensors and IoT boards) to the destination (Android Things app) without human action.

Let us describe the main components used in this project.

Project components

Before delving into the project details and how to use the MQTT protocol to exchange data, it is worthwhile to have an overview of the components and sensors used in this project:

- Wemos D1 mini (it acts as the *Sensors Unit Manager*):



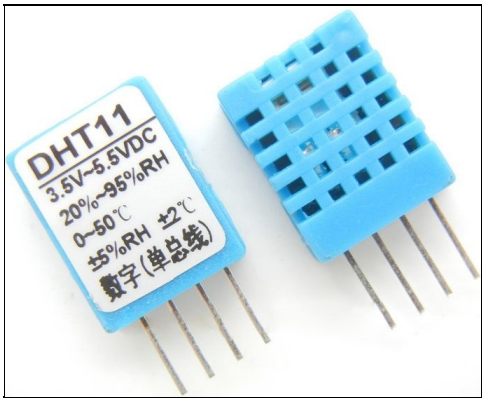
Source `https://www.wemos.cc/product/d1-mini.html`.

- This is compatible ESP8266 board that has a built-in Wi-Fi module. The project uses this board to manage a set of sensors to acquire data.

- Pressure sensor **BMP280.** We used it in Chapter 3, *How to Make an Environmental Monitoring System*:

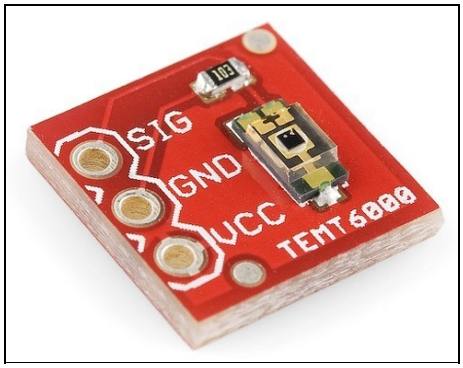
Source https://learn.adafruit.com/adafruit-bmp280-barometric-pressure-plus-temperature-sensor-breakout/overview

- The **DHT 11**, which that measures the temperature and humidity:



source http://www.electrodragon.com/product/humidity-and-temperature-sensor-dht11/#prettyPhoto/0/

- Finally, an ambient light sensor TEMT6000:



Sourcehttps://www.sparkfun.com/products/8688

In addition to these components, the project uses two other IoT boards:

- Arduino MKR1000 (https://www.arduino.cc/en/Main/ArduinoMKR1000)
- Raspberry Pi 2 (https://www.raspberrypi.org/products/raspberry-pi-2-model-b/)

The Arduino MKR1000 manages another set of sensors (*Sensors Unit 2*) that measure the same physical properties, except the light intensity. The Raspberry Pi 2 will act as a broker between the two sensors unit (Wemos and MKR1000) and Android Things board. We will cover these details in the next paragraphs.

The M2M architecture and the MQTT protocol

By now, in the previous chapters, we have covered how to acquire data from sensors connected to the Android Things board and how to use the Android Things board to manage remote IoT boards that are non-compatible with Android Things. Moreover, we use the Android Things board to send data to the cloud using IoT cloud platforms. There is another important aspect that is important to cover: how to use Android Things in **Machine to Machine** (**M2M**) architecture and the role the MQTT protocol plays in this architecture.

M2M is an emerging key component in Internet of things and **Industrial Internet of Things** (**IIoT**). M2M is focused on how machines talk to each other when exchanging data and information. In other words, with Machine to Machine terms, we refer to all the technologies and wireless networks that enable real-time data exchanging without human actions. In other words, the machine (or object) exchanges data by itself with other machines. This aspect is very important because it opens new application scenarios, such as:

- Telemetry
- Real-time failure notification
- Remote machine status control
- Real-time data acquisition

In this scenario, an important role is played by the *MQTT protocol*. It is important to know how it works so that we can exploit its features in our IoT projects. In the next paragraph, we will cover the MQTT protocol details before implementing it in our IoT project.

MQTT protocol overview

Message Queue Telemetry Transport (**MQTT**) is a light-weight message-based protocol. MQTT is an open protocol widely used in Internet of things in **Machine to Machine** (**M2M**) data exchange. It was developed around 1999 and now it is an OASIS standard (https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt). This protocol is easy to use and implement. It was designed with the target of having a small overhead. As we said before, MQTT is suitable for M2M communication where there are network bandwidth constraints. At the time of writing, the latest version of MQTT is 3.1. The open nature of this protocol and its features are powering the MQTT adoption. Moreover, there are several open-source implementations for different devices and platforms providing a wide range of options. Furthermore, several IoT cloud platforms have adopted it as a standard protocol to transfer information from IoT boards.

MQTT can be profitably used in several scenarios where message delivery is the main target while the network is unreliable. Generally speaking, Message Queue Telemetry Transport, even if it is largely adopted in the IoT ecosystem, it is not limited to it. There are other integration scenarios where MQTT features fit perfectly such as data exchange between smartphones, tablets, or other devices. Just to name a few common use cases where MQTT plays an important role, we can remember:

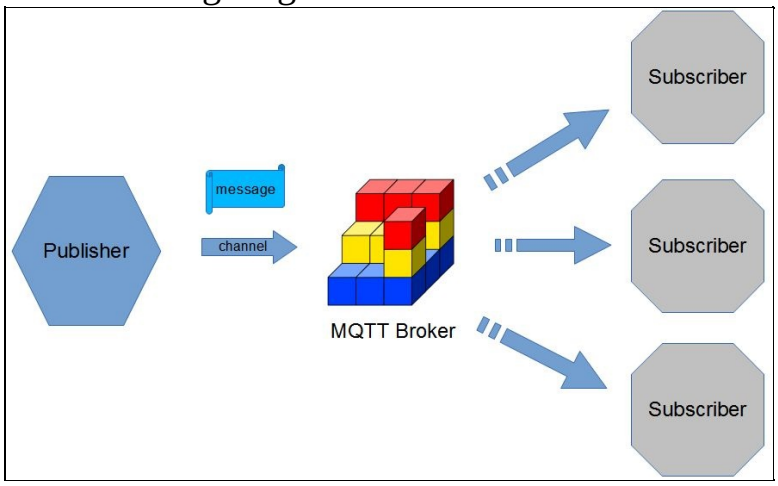
- Telemetry
- Notification systems
- Smart home

One important aspect to consider when using MQTT is that this protocol is a clear protocol. In other words, MQTT does not implement a security mechanism by itself.

We will cover the security aspects later.

MQTT message details

As stated before, MQTT is a message-centric protocol, or, in other words, the clients that participate in the data exchange process send and receive messages. MQTT is based on the *publish/subscriber* pattern. The following diagram shows the interactions between participants:

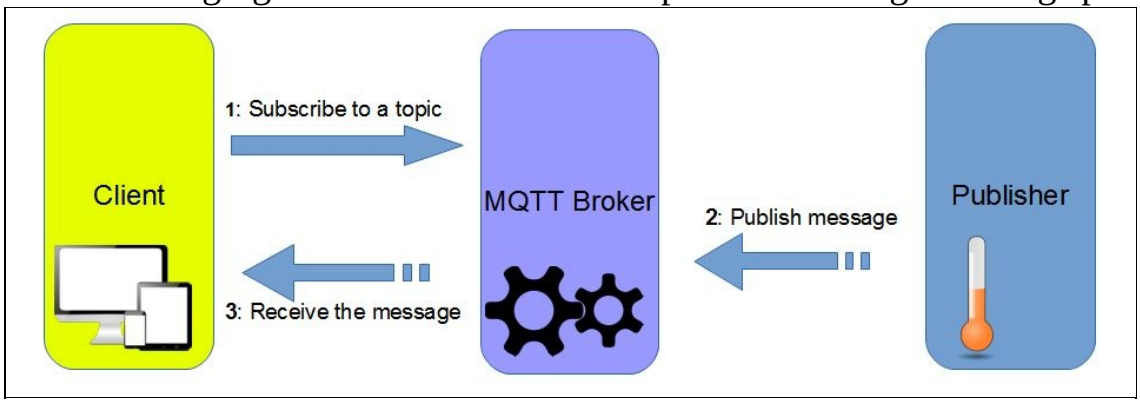


In MQTT there are three main participants in the message exchange process:

- **Publisher**: This is the device that produces the message. It is the source of the information. In our IoT project, we have at least two publishers. They are the IoT boards that acquire data from sensors and send it.
- **MQTT Broker**: This is a key component that enables the message flow from the publisher to the subscriber. It acts as a dispatcher, receiving messages from the publisher and forwarding them to the subscribers.
- **Subscriber**: This is the device/client that is interested in receiving messages from the publisher. In our IoT project, the Android Things board is the subscriber.

In order to filter the subscribers that will receive the message coming from the publisher, the MQTT broker uses a *topic*. A topic is a virtual channel between the publisher and the subscribers. In the MQTT context, a topic is represented as a UTF-8 string. Topics can be combined together in the same way we are used to managing folders and subfolders.

The following figure describes the main steps in the message exchange process:



These are the main steps:

1. A client subscribes to a topic declaring that it's willing to receive messages.
2. A publisher starts publishing messages on the same topic used by the subscriber.
3. The messages arrive at the MQTT broker, which in turn forwards them to the client.

It is very important that you understand how MQTT works before implementing it in our Remote weather station project.

The message architecture implemented by MQTT has several benefits:

- Its decouples the source of the information from the consumer. The subscriber and the publisher do not know each other and they use the broker to send and receive messages.
- Its decouples the publisher and the subscriber from the time. When the publisher sends a message the subscriber does not have to be active and connected at the same time. The message sending process and the receiving process are non-blocking operations.

Security and QoS

As said previously, MQTT does not have a built-in security mechanism; in other words, the message exchanged is in clear-text. To keep the footprint small, the MQTT relies on the existing security mechanisms and technologies. A common approach to this problem is using a secure transport layer. We can implement other mechanisms, such as message encryption and so on. If the scenario where the MQTT will be used has security requirements, it is

important to adopt one of the previous mechanisms.

The last aspect is related to the *Quality of Service* (QoS). MQTT supports three level of QoS:

- **At most once (QoS 0)**: A message is delivered at most once or is not delivered
- **At least once (QoS 1)**: A message is delivered at least once. If the receiver do not send the acknowledgement message, the message is sent again
- **Exactly only one (QoS 2)**: A message is delivered once and only one time

The QoS plays an important role in MQTT because it frees the publisher and the subscriber from the need to handle network problems. In other words, it is the protocol that handles retransmission attempts when there are network failures and it guarantees that the message is delivered.

Using MQTT in our remote weather station

Now it is time to delve into the project details and explore how sensors and IoT boards exchange data with the Android Things board. The project has two parts:

- IoT boards that manage sensors and acquire data
- The Android Things board, which collects data

In the MQTT architecture model, the IoT boards act as *publishers* that publish sensor data to an MQTT channel while the Android Things board is a *subscriber* that receives the data published by the IoT boards. Every IoT board used in this project publishes data using a specific channel so that the Android Things app can know where the data comes from.

It is important to have a clear view of the components and how we will implement them:

Component	**MQTT Role**
Wemos D1 (ESP 8266)	Publisher (topic:channel1)
Arduino MKR1000	Publisher (topic:channel2)
Raspberry Pi2	Server
Raspberry Pi3/Intel Edison (Android Things)	Subscriber

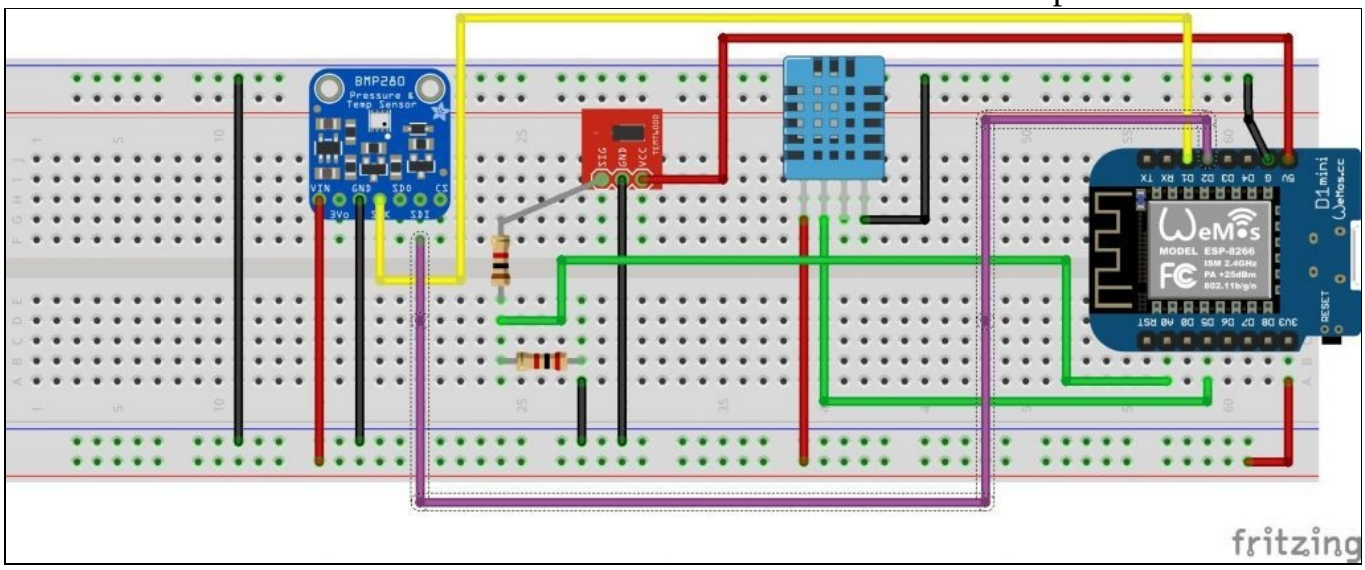
This chapter will cover how to install the MQTT server on Raspberry Pi2 at the end of this chapter. For now, it is enough to know that this project uses the Mosquitto server (http s://www.eclipse.org/mosquitto/download/).

Implementing the MQTT publisher

In this first step, we use the *Wemos D1 mini* (or any compatible ESP8266 board) to read data from sensors and publish it to an MQTT topic. The physical properties measured by this IoT project are:

- Humidity
- Temperature
- Pressure
- Light

It is worthwhile to know how to connect the sensors described in the previous sections:



The connections are very simple:

- DHT11 has its data pin connected to the D5 pin, while the Vcc is connected to +3.3V
- The BMP280 that we use to measure the pressure is an *I2C sensor*, so it has four connections:
  - Vcc is connected to +3.3V
  - The clock signal is connected to D1 pin
  - The data signal is connected to D2 pin
  - The GND is connected to the common ground
- The TEMT6000 is an analogic sensor and it is powered using +5V. At the output, we use a voltage divider so that the output signal is always lower than 3.3V

To develop the sketch we need an Arduino IDE configured with Wemos support. To know more follow this guide (https://www.wemos.cc/tutori al/get-started-arduino.html).

Once your IDE is configured, we can start coding:

1. Open a new sketch and add the following lines:

```
#include "Adafruit_Sensor.h"
        #include <DHT.h>
        #include <ESP8266WiFi.h>
        #include <Adafruit_BMP280.h>
```

Notice that this project requires you to install the libraries to manage the sensors described previously. For this purpose, you can use the Arduino library manager.

2. Add the following constants that hold your Wi-Fi configuration parameters:

```
const char* ssid = "your_SSID";
        const char* pwd = "your WiFi password";
```

3. Now we have to define the DHT type (DHT11 in this project) and the pins used to connect the sensors. At the end of these lines, we define the pin used to connect to the ambient light sensor:

```
#define DHTPIN D5
        #define DHTTYPE DHT11
        #define TEMTPIN A0
```

4. Now we initialize the sketch and configure the WiFi connection:

```
WiFiClient client;
        DHT dht(DHTPIN, DHTTYPE);
        Adafruit_BMP280 bme; void setup() {
        Serial.begin(115200); dht.begin();
        WiFi.begin(ssid, pwd); Serial.println("Connecting
        to Wifi...");
        while (WiFi.status() != WL_CONNECTED) {
         Serial.println(".."); delay(400);
         }
        bme.begin();
        Serial.println("Wifi connected.");
        }
```

5. Now we can read the data from sensors:

```
void loop() {
        float h = dht.readHumidity();
        float t = dht.readTemperature();
        float press = bme.readPressure() / 100;
        int lightVal = analogRead(TEMTPIN) * 0.9765625;
        delay(5000);
        }
```

Now, we have developed the piece of the sketch that handles sensors and reads data from them. The next step is sending this information through an MQTT protocol.

Connecting to MQTT and sending data

In this step, we connect to the MQTT server and start publishing data to the MQTT topic. This module uses a topic named *channel1*. The data is published using JSON format so that the MQTT subscriber can retrieve and parse it easily. To use the MQTT protocol, we have to import a library that simplifies the development process. There are several MQTT libraries available and you can select one of them. Almost all work in the same way, so you can easily switch between them. In this project, we use `PubSubClient`, therefore you have to import it before using the library in our project.

Now, we have to transform the sketch described previously, adding the publishing feature. To do this, follow these steps:

1. At the beginning add the following line:

```
#include <PubSubClient.h>
```

2. Now we have to set up the MQTT publisher:

```
PubSubClient mqttClient(client);
```

   Notice that `mqttClient` accepts as parameter the client that handles the Wi-Fi connection.

3. Once we have defined the client, we have to configure the connection details, server address, and the port. Therefore add the following lines:

```
void initMQTT() {
          mqttClient.setServer(mqtt_server, 1883);
        }
```

   We invoke `initMQTT()` in the setup method.

4. Finally, it is time to publish the message. We want to publish the data using the JSON format so that the subscriber can parse it easily. In the `loop()` method, after the lines that read data from sensors add the following lines:

```
String payload="{"temp":"" + String(t) + "",
      "hum":"" + String(h) + "", "press": "" +
      String(press) +"", "light":"" + String(lightVal)
      + ""}";
      mqttClient.publish(topic, payload.c_str());
```

   Notice that the sketch publishes the data through a topic. In this case, the topic is *channel1*.

Congratulations! You have just implemented your first MQTT publisher ready to send data.

This project uses another MQTT publisher based on MKR1000. You can find the sketch in the companion source code.

repositories { jcenter() mavenCentral()
 }

private MQTTClient(Context ctx) {
 this.ctx = ctx;
 }

public static final MQTTClient getInstance(Context <br/> ctx) {
 if (me == null) me = new MQTTClient(ctx);
 return me;
 }

public void connectToMQTT() {
 ...
 }

String clientId = MqttClient.generateClientId();
 mqttClient = new MqttAndroidClient(ctx,
 MQTT_SERVER, clientId);

try {
 IMqttToken mqttToken = mqttClient.connect();
 mqttToken.setActionCallback(new
 IMqttActionListener() {
 @Override public void onSuccess(IMqttToken <br/> asyncActionToken) {
 Log.i(TAG, "Connected to MQTT server"); }
 @Override
 public void onFailure(IMqttToken asyncActionToken,
 Throwable exception) {
 Log.e(TAG, "Failure");
 exception.printStackTrace();
 }});
 }
 catch (MqttException mqe) {
 Log.e(TAG, "Unable to connect to MQTT Server");
 mqe.printStackTrace();
 }

public void subscribe(final String topic) {
 try { IMqttToken subToken =
 mqttClient.subscribe(topic, 1);
 subToken.setActionCallback(new
 IMqttActionListener() {
 @Override public void onSuccess(IMqttToken <br/> asyncActionToken) {
 Log.d(TAG, "Subscribed to topic ["+topic+"]");
 }
 @Override public void onFailure(IMqttToken <br/> asyncActionToken,Throwable exception) {
 Log.e(TAG, "Error while subscribing to the <br/> topic ["+topic+"]");
 exception.printStackTrace();
 }
 });
 // Subscribe to other topic
 } catch (MqttException e) {e.printStackTrace();
 }
 }

public class MQTTClient implements MqttCallback {
 ...
 }

@Override
 public void connectionLost(Throwable cause) {
 }
 @Override
 public void deliveryComplete(IMqttDeliveryToken <br/> token) {
 }

15. By now, we have just overridden them with an empty implementation. Anyway, in the first method `connectionLost`, invoked when the connection is not available, we can use this method to try to reconnect to the server

16. The `MQTTClient` class, in turn, has to expose a set of callback methods to inform the caller about the MQTT events. To do it, we can create a simple interface with the methods related to the event we want to notify:

```
public interface MQTTListener { public void onConnected();
public void onConnectionFailure(Throwable t);
public void onMessage(String topic, MqttMessage message); public void c
}
```

17. That's all. The MQTT subscriber handler class is ready.

private void initMQTT() {
 mqttClient = MQTTClient.getInstance(this);
 mqttClient.setListener(this);
 mqttClient.connectToMQTT();
 }

 public class MainActivity extends Activity <br/> implements
 MQTTClient.MQTTListener {
 ...
 }

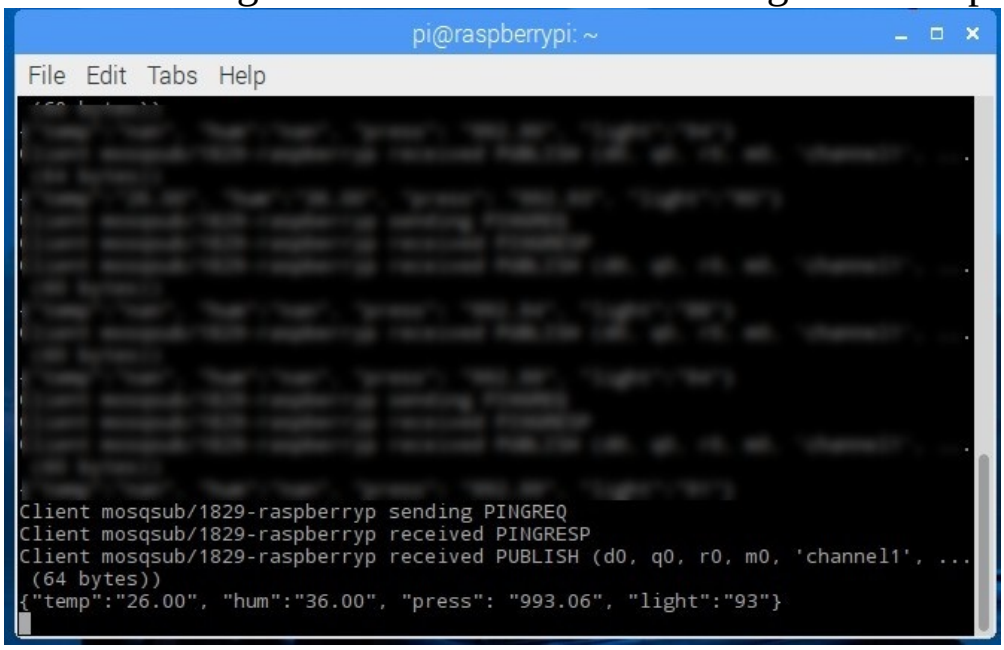
@Override
 public void onConnected() {
 mqttClient.subscribe("channel1");
 mqttClient.subscribe("channel2");
 }

@Override
 public void onMessage(String topic, MqttMessage <br/> message) {
 // Extract the message and update the view
 }

 mosquitto_sub -d -t channel1

3. In this way, we subscribe to the same channel used by Android Things to receive messages. Now wait for the publisher to start sending messages. You will see the messages arriving on the server.

The following screenshot shows the message showed previously in Android Things UI:

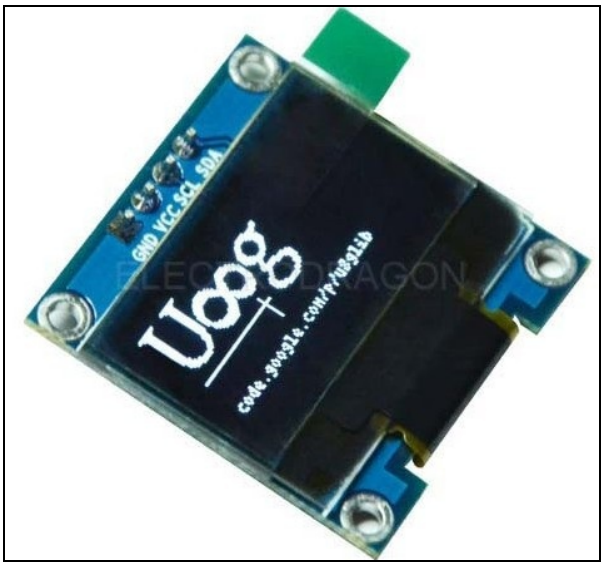


You can notice the JSON message coming from the publisher.

Displaying the information using OLED display

By now you have used an Android Things UI to display the information coming from publishers. Anyway, as we noticed in the previous chapters, not all the Android Things compatible boards support a user interface. For this reason, we have to use a different approach to displaying the information. In Chapter 5, *Create a Smart System to Control Ambient Light*, we showed how to implement a simple Web server to expose a Web interface to interact with the Android Things app. In this project, we do not need to interact with the app user interface, but the Android Things app simply has to show the result. To this purpose, we can use an OLED display. As you may already know, OLED stands for *organic light- emitting diode*.

This display can be connected to the Android Things board and the app can control it. The display we will use in this project is quite small, but you are free to use a wider display. The following image shows the SSD1306 OLED display:

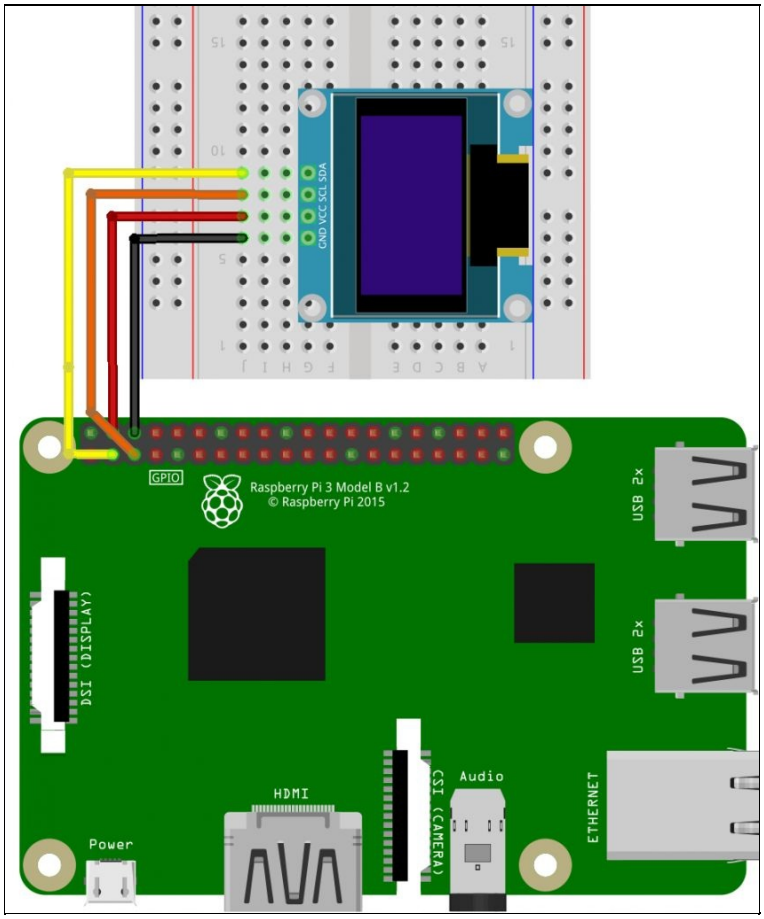


Source http://www.electrodragon.com/product/0-96-12864-oled-display-iicspi/?attribute_pa_interface=iic#prettyPhoto[product-gallery]/5/

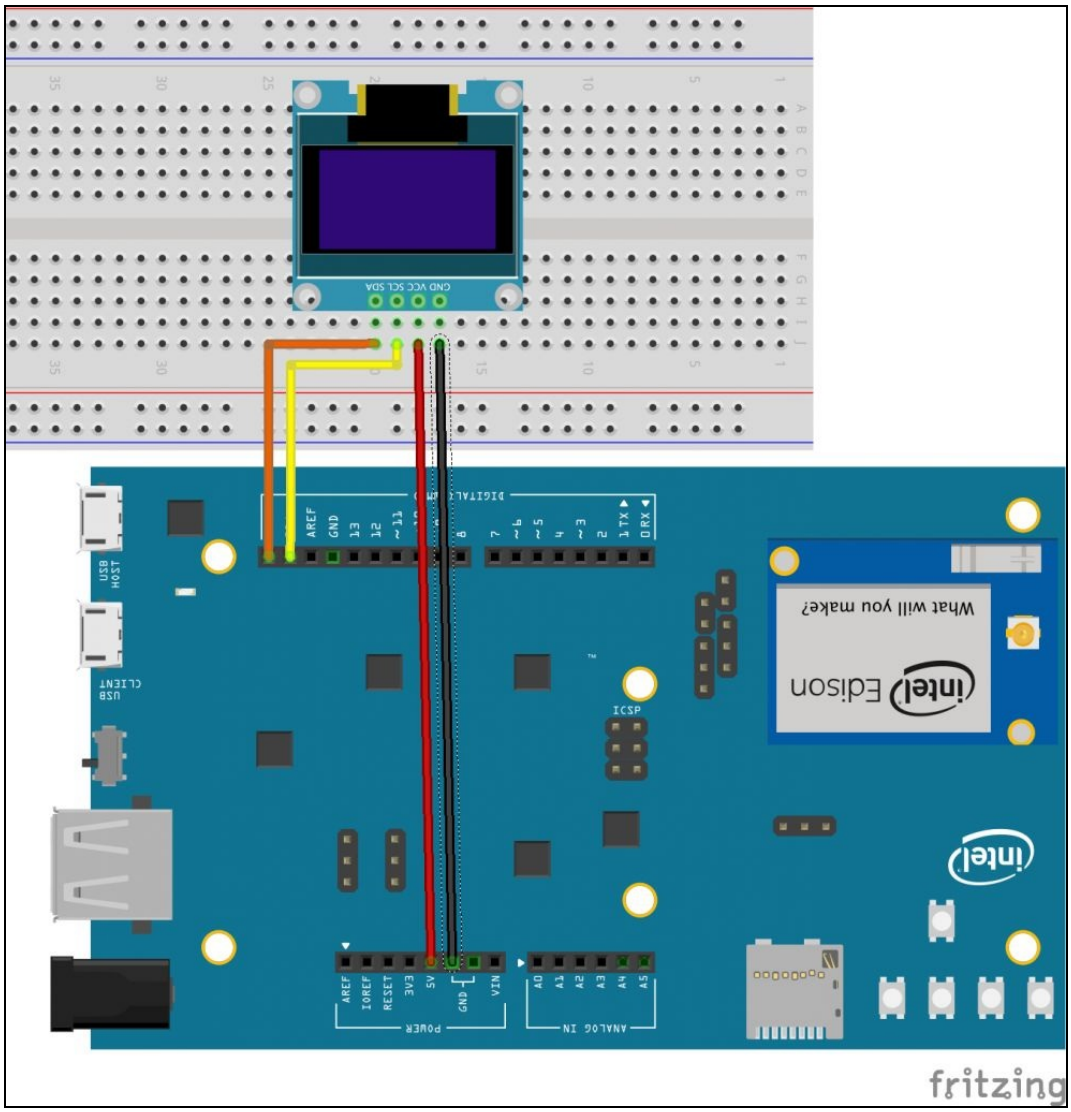
This display uses the I2C protocol covered in Chapter 3, *How to Make an Environmental Monitoring System*.

Connect the OLED display to Android Things board

The first step is connecting the OLED display to Android Things board. In this project, we have used Raspberry Pi3 and Intel Edison with Arduino breakout kit. The following figure describes how to wire the OLED I2C display to Raspberry Pi3:



While the following figure shows how to wire the OLED I2C display to Internet Edison with Arduino breakout kit:



Once we know how to connect it we can start using this peripheral:

1. The first step is importing the library into our project declaring the dependency in the `build.gradle` file at app level. To this purpose open this file and add the following lines inside the dependencies directive:

```
        \compile 'com.google.android.things.
        contrib:driver-ssd1306:0.2'
```

2. Now we are ready to use the peripheral. Let us create a new class that takes care of writing the information on the display. We will name this class `DisplayManager.java`:
3. Add the following lines to the previous class:

```
private Ssd1306 display;
        private Handler h = new Handler();
        public DisplayManager() {
        try {
        display = new Ssd1306(getI2CPin());
        display.clearPixels();
        }
        catch(Exception e) {
        e.printStackTrace();
        }
        }
```

4. Notice that we initialize the instance of the class that will manage the display using the *I2C SDA* pin

name. As you may already remember, the pin name changes according to the Android Things board. For this reason, in the preceding code we use the `getI2CPin()` method that returns the name of the pin according to the board type.

5. Now we have to implement the method that writes the message on the display. If you have a look at the `Ssd1306` class, it has only a few methods that perform some basic tasks, such as:
   - Writing a pixel
   - Turning on and off the display
   - Scrolling
   - Clear the screen
   - Getting the width and the height of the screen

6. We cannot show the message on the display turning on the pixels manually. Fortunately, there is a class called `BitmapHelper` that helps us to represent an image on the display. The idea then is using the Android Things graphics API to create a bitmap that holds the message we want to write and then send it to the display. Therefore, the first step is creating a method that creates the bitmap. We call it `displayMessage` with this content:

```
int width = display.getLcdWidth();
        int height = display.getLcdHeight();
        Bitmap b = Bitmap.createBitmap(width, height,
        Bitmap.Config.ARGB_8888);
```

7. Now we configure the `Paint` class that takes care of the style and the color of the message we are displaying:

```
Paint p = new Paint(Paint.ANTI_ALIAS_FLAG);
        p.setTextSize(size);p.setColor(color)
        p.setTextAlign(Paint.Align.LEFT);
```

   Where `size` and `color` are the method input parameters. Moreover, we want the message aligned to the left.

8. The next step is creating a `Canvas` (https://developer.android.com/reference/android/graphics/Canvas.html), that as stated in the Android `javadoc`, `holds` the draw:

```
Canvas c = new Canvas(b);
```

   Where `b` is the `Bitmap` we have configured in step 5. Now we write the message using the style and the color defined previously (step 6):

```
c.drawText(msg, 0, 0.5f * height, p);
```

9. Finally, we invoke the `BitmapHelper` passing the bitmap we have just created holding the message we want to visualize:

```
BitmapHelper.setBmpData(display, 0, 0, b, true);
```

10. The last thing to consider is that we do not want that this process could block the app while it is creating the bitmap. Therefore, we use a `Runnable` class that wraps all the steps described previously. To do it, let us modify the `displayMessage()` method in this way:

```
public void displayMessage(final String msg, final
```

```
        int size,
        final int color) {
          Runnable r = new Runnable() {
            @Override public void run() {
              // All the code used above to display the
              message
            };
          h.post(r);
         }
```

Here, `h` is is an instance of the `Handler` class.

Installing the MQTT server

The last step of this wide IoT project is installing the MQTT server. As you already know, this is the link between the publisher and the subscriber. We added this step at the end because it is not the main focus of this chapter, but it is important to know how to do it so that you gain a deep knowledge about MQTT and the steps necessary to implement an IoT ecosystem that exchanges data and information using MQTT protocol. As stated before, this IoT project uses Mosquitto an open source MQTT broker (https://mosquitto.org/). There are several versions working on different operating systems. You can find more information about the platforms supported at this link (https://mosquitto.org/download/). In this project, we will use the Raspberry Pi 2, anyway, you are free to use any version you like according to your needs.

Installing the MQTT broker

To install the MQTT broker you have to connect to Raspberry Pi 2 and follow these steps:

1. The first step is adding a repository that holds the application. Before doing it, it is necessary to add the key to authenticate the repository:

```
wget http://repo.mosquitto.org/debian/mosquitto-
repo.gpg.key
```

2. Now we have to import the key:

```
sudo apt-key add mosquitto-repo.gpg.key
```

3. The last step to configure the repository is adding the file: .list

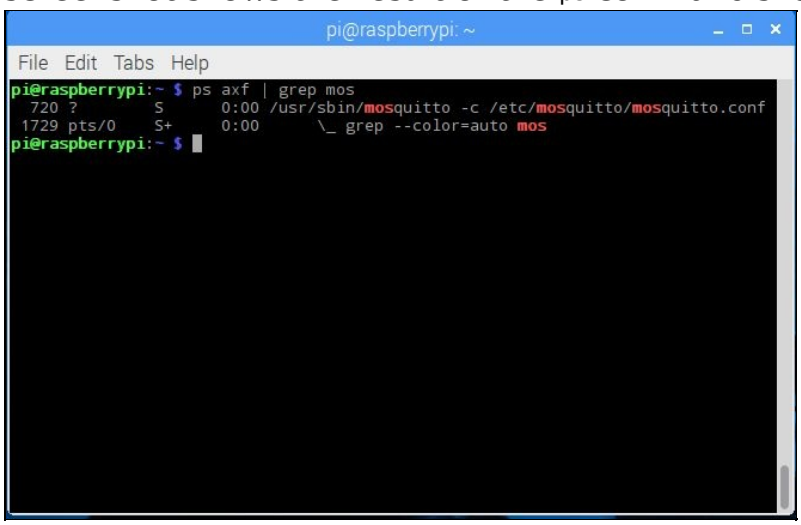
```
sudo wget
http://repo.mosquitto.org/debian/mosquitto-
wheezy.list
```

4. Now the repository is configured and we can download and install the application:

```
apt-get install mosquitto
```

The installation is complete now. You can start using the MQTT broker connecting the devices described previously. Moreover, to complete the installation, we can add the client library so that we can test the installation or create a subscriber/publisher. To install the client follow this step: apt-get install mosquitto-clients

Now you can test the installation to verify that everything works correctly. The following screenshot shows the result of the `ps` command showing the mosquitto process up and running:



You can notice that the mosquitto uses the configuration file that we will describe in the next paragraph.

Configuring the MQTT broker

The installation we have just completed uses default parameters. You can configure the MQTT broker according to your needs. To customize the broker you have to use `mosquitto.conf`, that is the configuration file. Here you can change several parameters; you can find the details at this link (https://mosquitto.org/man/mosquitto-conf-5.html).

You can, for example, change the IP address used by the server to listen to incoming connections or the port. These two properties are:

- `bind_address address`: It is used to set the IP address where the server listens for the incoming connection
- `listener port`: It is port used to listen to

Summary

At the end of this chapter, you have gained the knowledge about MQTT protocol. We have explored how to build a complex IoT system that uses heterogeneous components. We have explored how to build a remote weather station that acquires data remotely and send it through MQTT to Android Things board. Using this project, we have covered an important topic related to M2M. In the next chapter, we will explore how to use **Pulse Width Modulation** (**PWM**) to control servo motor using Android Things and how to acquire images using cameras.

Build a Spying Eye

In this chapter, we will build a spying eye. In more detail, we will build a project that uses an Android Things board to control a camera and a servomotor that we will use to rotate the camera. Through this project, we will learn how to use *PWM* pins. TheÂ **Pulse Width Modulation** (**PWM**) pin is a different kind of pin that we will use to control different types of peripherals such as servo motors. In more details, in this chapter we will explore:

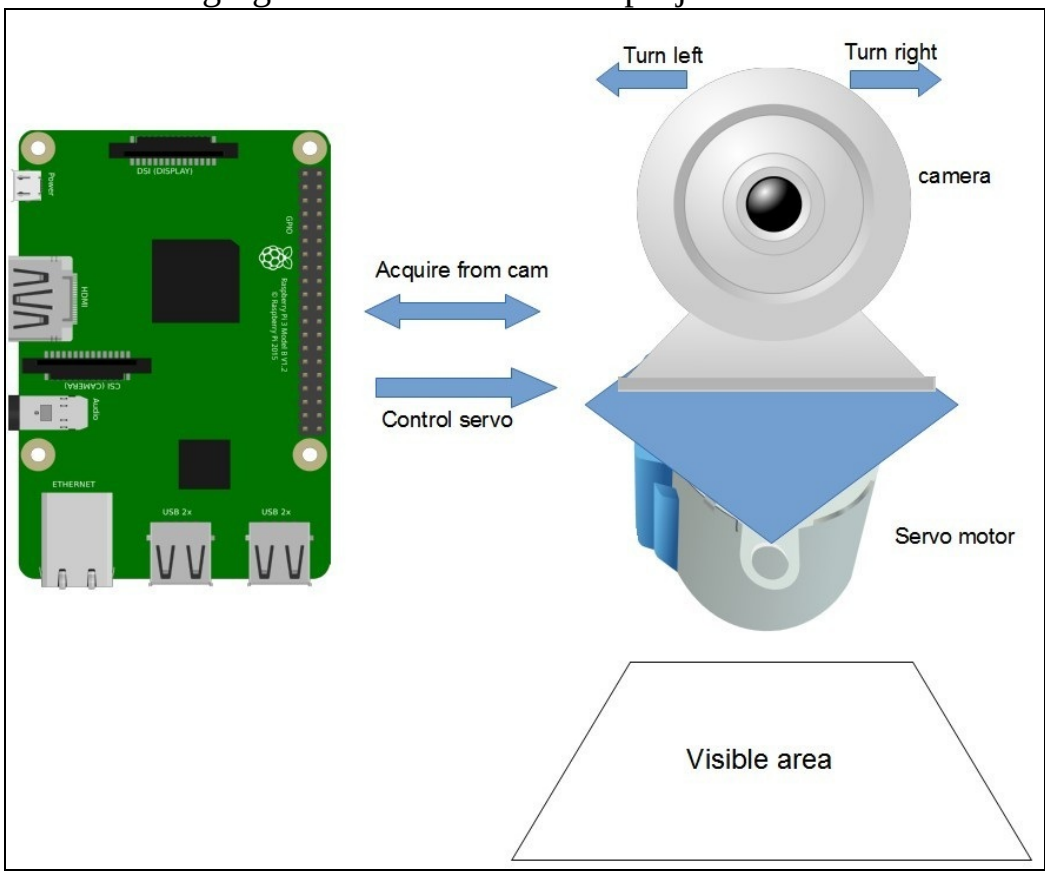
- How to use PWM pins in Android Things
- How to control servo motors
- How to use cameras

At the end of this chapter, we will build a fully working Android Things system that is able to acquire images using the camera and, at the same time, we will be able to control the camera direction using a servo motor.

Spying eye Android Things project overview

Before digging into the project details, it is worthwhile to have an overview of this project to know what we want to build and how it should work. The basic idea that stands behind this project is using a servo motor at the camera base. The camera and the motor are connected together so that while the Android Things app rotates the motor, we can change the camera orientation. By the way, a servomotor is a special motor type that we can control precisely in terms of its angular position.

The following figure visualizes the main project features:



As you can see from the preceding figure, in this project we want the Android Things board controlling the following at the same time:

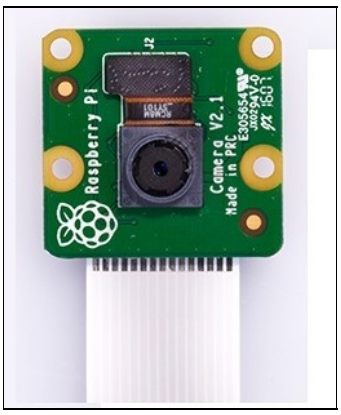
- The servomotor rotation
- The camera that acquires the image

To access these features, this project exposes a simple UI interface. An important aspect is related to the camera support. Android Things supports CSI-2 protocol. There are some Android Things compatible boards that do not support a camera. For example, Intel Edison with breakout kit does not support CSI-2, so we cannot connect the camera. For these reasons, for the first time in this book, we will use only Raspberry Pi 3.

Project components

The components used in this project are:

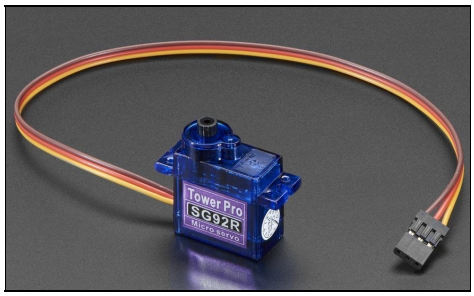
- Raspberry camera module:



Source: https://www.raspberrypi.org/products/camera-module-v2/

This camera is based on the Sony IMX219 8-megapixel sensor. It can be used to take high-definition videos or pictures.

- A servomotor. There are several types of servomotors with different specifications. You can use for example:



Source https://www.adafruit.com/product/169

- The following is an optional component that we will use to hold the camera and attach it to the motor:

Sourcehttps://www.amazon.it/gp/product/B00IJZJKK4/ref=oh_aui_detailpage_o00_s00?ie=UTF8&psc=1

Now we know the component that we will use in this Android Things project, it is useful to have an overview about PWM and how to use it in Android Things so that we get confident with servo motors.

Pulse Width Modulation overview

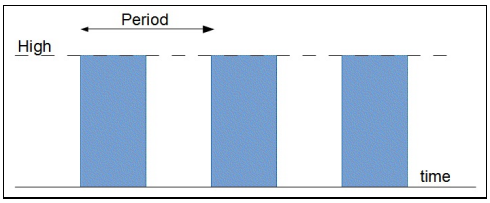
Until now, we have seen different ways to control and exchange data with external peripherals. There is another method that we can use to control external devices. This is called **Pulse With Modulation** (**PWM**). It is a modulation technique widely used in several fields; anyway, one of the most interesting applications is controlling the power supplied to an external device. In other words, PWM is a technique that enables us to create a variable voltage using a digital signal. We can use PWM to control:

- Servomotors
- Light intensity (in a LED)
- Sound and audio

This technique is based on changing the time when the signal is high. There are two important factors in this modulation:

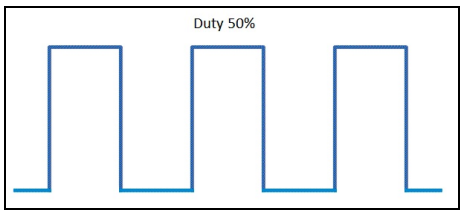
- Frequency
- Duty cycle

We all know the frequency; we can define it as the number of occurrences of a repeating event per unit time. Related to the frequency, there is the **Period** that is the duration time of one cycle.



On the other hand, the duty cycle is the amount of the time when the signal is high with respect to the signal period. The duty cycle is measured as a percentage. The percentage duty cycle represents the percentage of the time when the signal is high. The following is an example to help you to better understand this concept:

If the signal is half the time high and the other half low we say that the duty cycle is 50%:



- If the signal is always high, the duty cycle is 100%

It is important to know these parameters because they are used by Android Things to handle PWM signals.

PeripheralServiceManager psm = new
 PeripheralServiceManager();

pwmPin = psm.openPwm(getBoardPin());

pwm.setPwmFrequencyHz(50);
pwm.setPwmDutyCycle(75);

pwm.setEnabled(true);

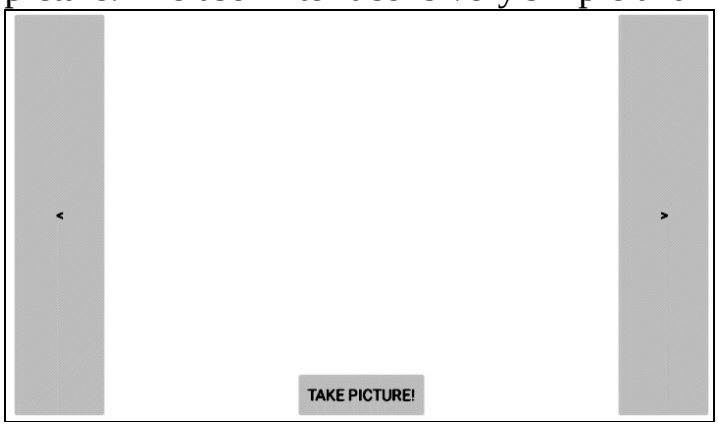
Do not forget to close the pin when the Activity is destroyed as we did for other kinds of pins.

Implementing the spying eye project in Android Things

It is time that we focused our attention on developing the project. Now we have all the information and the knowledge we need to build the Android Things app. This app can be divided into two different parts:

- Control servomotor
- Use camera in Android Things

In the first part, we describe, according to the information explained previously, how to control a servomotor while, in the second part, we will describe how to acquire images in Android Things using a camera. As you may remember, the servomotor is used in this project to rotate the camera so that we can explore a wider area. Moreover, this project has a UI that we can use to control the servo and take the picture. The user interface is very simple and intuitive; the result is shown in the following figure:



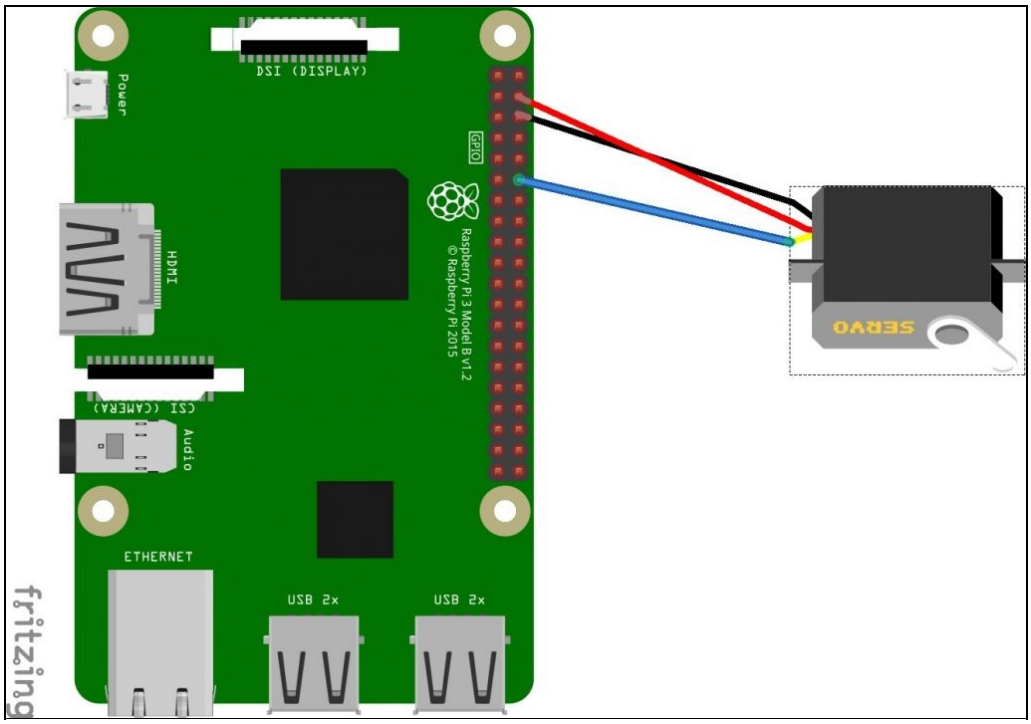
Basically, there are three buttons:

- The left button is used to turn the camera to left rotating the servo
- The right button is used to turn the camera to right rotating the servo
- The button in the middle is used to take the picture

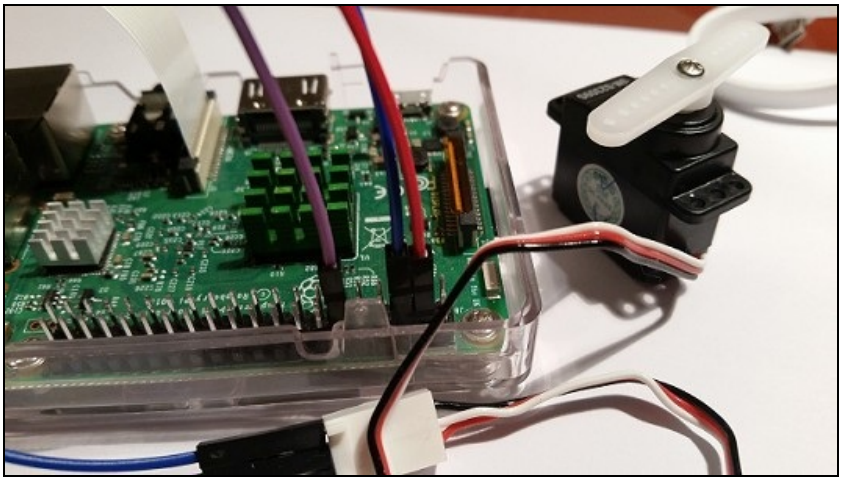
Let us see how to develop it.

Controlling a servomotor in Android Things

Before using the PWM it is important to know how to connect a servo motor to an Android Things board. In this project, we will use only Raspberry Pi 3 because it supports a camera; anyway, you can connect the servo to Intel Edison with Arduino breakout too. The following diagram describes the connections:



The following is an image of the Raspberry Pi3 connected to the servo:



Usually, a servo has only three wires:

- The power signal that depends on the type of motor (usually between 3.3V and 5V)
- The ground signal
- The control signal that we have to connect to the PWM pin

A servo motor is controlled using a PWM signal so we can control it applying the steps described previously. Anyway, Android Things provides a library that helps us to control a servo motor more easily. We need to know about the servo so we can control its rotation. This library provides a set of

methods to deal with angles instead of frequencies and duty cycles so that it gets easier to precisely control the servo position. In this Android Things project, we will use this library to simplify our work.

Let us create a new project as described in Chapter 1, *Getting Started with Android Things*, cloning the reference project. This project, as stated before, has a UI, but we will not describe its layout structure. You can refer to the companion code of this book to know how to code the layout. Let us focus our attention on the servo. To control it, follow these steps:

1. Open the `build.gradleat` app level and add the dependency to the library described previously:

```
dependencies {
        provided
         'com.google.android.things:androidthings:0.3-
         devpreview'
         compile
        'com.google.android.things.contrib:driver-
         pwmservo:0.1'
        }
```

2. Now open the `MainActivity.java` and add the following lines:

```
private void initServo() {
      try {
           mServo = new Servo("PWM0");
           mServo.setAngleRange(0f, 180f);
           mServo.setEnabled(true);
          }
           catch(Exception e) {
           e.printStackTrace();
           }
          }
```

   In the preceding code, we have hard coded the pin name because this project will work only on Raspberry Pi3. You have to change the pin name if you run this Android Things app using another board. Moreover, we set the minimum and the maximum angle. These values define the rotation angle range. Finally, we enable the pin.

3. In the `onCreate` method, we have to call `initServo`, so add the following line:

```
initServo();
```

4. Now we have to handle the two buttons to control the servo rotation. At the beginning we get the reference to the buttons:

```
Button btnLeft = (Button)
        findViewById(R.id.btnLeft);
        Button btnRight = (Button)
        findViewById(R.id.btnRight);
```

5. Then we have to handle the click events and rotate the servo accordingly:

```
btnLeft.setOnClickListener(new
        View.OnClickListener() {
        @Override
        public void onClick(View v) {
        angle += STEP;
        setServoAngle(angle);
        }
        });
        btnRight.setOnClickListener(new
        View.OnClickListener() {
```

```
        @Override
        public void onClick(View v) {
        angle -= STEP;
        setServoAngle(angle);
        }
        });
```

Here, the `STEP` is the step we want to rotate the servo.

6. Finally, we have to define the `setServoAngle` method that actually rotates the servo:

```
private void setServoAngle(int angle) {
        if (angle > mServo.getMaximumAngle())
          angle = (int) mServo.getMinimumAngle();
        if (angle < mServo.getMinimumAngle())
          angle = (int) mServo.getMinimumAngle();
        try {
          mServo.setAngle(angle);
        }
        catch (IOException e) {
         e.printStackTrace();
         } }
```

In this method, we control that the rotation angle is between the minimum and maximum values; otherwise, we set the limit angles. Finally, we use `setAngle` to set the rotation angle on the servo motor.

That's all. You can run the Android Things app and test it using the UI. When you click on the left or right button, the servo motor has to rotate in the correct direction.

The first part is completed: now we can control the servo motor using the Android Things UI. In the next paragraph, we will take a picture using the camera.

Using a camera in Android Things

In this section, we will cover a new aspect of Internet of Things: how to use a camera. Until now, we have connected to the Android Things board several devices using GPIO pins. A camera is different from all the peripherals covered previously and we use a different way to connect it. Not all the Android Things compatible boards support an external camera. At the time of writing, only these boards support it:

- Raspberry Pi3
- Intel Joule

The camera is connected using a **Common Serial Interface** *(***CSI-2**). You have to use a compatible camera as specified at the beginning of this chapter. In order to handle the camera, we will use `android.hardware.camera2` (added from API level 21). This package provides all the classes and interfaces necessary to handle a camera connected to an Android device. As we will see later, the process to take a picture in Android Things is the same used in Android. In this package there are some important classes that are at the heart of the code we will implement later:

- `CameraManager`: This class represents a system manager that we will use to detect the connected camera and to open it (https://developer.android.com/reference/android/hardware/camera2/CameraManager.html)
- `CameraDevice`: This class represents the camera connected to our device in terms of properties and capabilities (https://developer.android.com/reference/android/hardware/camera2/CameraDevice.html)
- `CaptureSession`: This represents the method we use to capture the image and represent it on a surface

Let us explore how to implement this second project part. Considering that the camera management is quite complex, it is useful to implement all the steps necessary in another class:

1. Open the project used until now to control the servo.
2. Add another class named `AndroidCamera.java` to the project. This class will handle all the details related to the camera.

In the next paragraph, we will analyze step by step how to use the camera.

Getting ready to use the camera

Before using the camera, we have to detect it and make sure it is connected to the Android Things board. For this purpose, we will use the `CameraManager` class:

1. In `AndroidCamera.java` let us create a new method called `initCamera()`.

2. In this method, we first get the reference to the camera manager and then the app enumerates all the connected cameras:

```
cManager = (CameraManager)
        ctx.getSystemService(Context.CAMERA_SERVICE);
            try {
            String[] idCams = cManager.getCameraIdList();
            camId = idCams[0];
            }
           catch (CameraAccessException e) {
           e.printStackTrace();
           }
```

   In this project, we will use the first camera detected (as you can notice we use *0* as index).

3. Moreover, in this method, we have to initialize an image container that is used by the app to direct access to the data rendered into a surface:

```
imgHandler.start();
        iReader = ImageReader.newInstance(320, 240,
        ImageFormat.JPEG, 1);
        iReader.setOnImageAvailableListener(new
        ImageReader.OnImageAvailableListener() {
          @Override
           public void onImageAvailable(ImageReader
        reader) {
             listener.onImageReady(reader);
         }
       }, new Handler(imgHandler.getLooper()));
```

   The preceding code is very simple; at the first line, the app initializes a `Handler` that is required by the `ImageReader`. Next, we create an `ImageReader` instance setting the *width* and the *height* and the image format. In this project, the `ImageReader` holds only one image. Finally, we attach a listener so that this class gets notified when the image is available. In turn, the `AndroidCamera` class uses another listener to notify to the caller (`MainActivity.java`) that the image is available.

4. The next step is implementing a method that is used to open the camera communication:

```
public void openCamera() {
        try {
        cManager.openCamera(camId, stateCallback, null);
        }
        catch (CameraAccessException e) {
          e.printStackTrace();
           }
        catch (SecurityException se) {
          se.printStackTrace();
         }
        }
```

As you can see, to open the camera we use the `camId` retrieved in the first step. Moreover, we use a `callback` class to be notified when the events related to this process happen.

5. Now we have to implement the `callback` class used in the previous step:

```
private final CameraDevice.StateCallback
        stateCallback = new
        CameraDevice.StateCallback() {
          @Override
          public void onOpened(@NonNull CameraDevice
            camera) { Log.d(TAG, "Camera opened");
            AndroidCamera.this.camera = camera;
            listener.onCameraAvailable();
         }
          @Override
          public void onDisconnected(@NonNull CameraDevice
         camera) { Log.d(TAG, "Camera disconnected");
         }
          @Override
         public void onError(@NonNull CameraDevice camera,
         int error) { Log.d(TAG, "Camera Error" + error);
         }
         };
```

There are several methods that we have to implement in the `callback` class. We are interested in the method invoked when the camera is opened, because we store the instance of the `CameraDevice` to refer to the connected camera in the next steps. At the same time, in the same method, we inform the caller that the camera is connected.

6. Finally, once the camera is connected, we can implement the method to take the picture:

```
public void takePicture() {
        try {
        camera.createCaptureSession(
        Collections.singletonList(iReader.getSurface()),
        sessionCallback, null);
        }
         catch(Exception e) {
        e.printStackTrace();
         }
        }
```

The app creates a *capture session* that is used to take the picture. As we did previously, we use a `callback` class to be informed about the events. Notice that the `createCaptureSession` method uses the `ImageReader` surface to hold the picture.

7. The `callback` method to handle *capture session events* is shown here:

```
private CameraCaptureSession.StateCallback
        sessionCallback =
        new CameraCaptureSession.StateCallback() {
        @Override
        public void onConfigured(
           @NonNull CameraCaptureSession session) {
           Log.d(TAG, "Camera configured");
           AndroidCamera.this.session = session;
           startCaptureImage();
          }
        @Override
        public void onConfigureFailed(
          @NonNull CameraCaptureSession session) {
          Log.e(TAG, "Configuration failed");
           }
         };
```

It is important to notice the `onConfigured` method that is called when the camera is ready to capture the image and the configuration process is finished. The app uses this method to start capturing the picture.

8. The last step is implementing the method that actually captures the image:

```
private void startCaptureImage() {
        try {
            CaptureRequest.Builder captureBuilder =
        camera.createCaptureRequest(
                CameraDevice.TEMPLATE_STILL_CAPTURE);
           captureBuilder.addTarget(iReader.getSurface());
           captureBuilder.set(
               CaptureRequest.CONTROL_AE_MODE,
               CaptureRequest.CONTROL_AE_MODE_ON);
            Log.d(TAG, "Session initialized.");
         session.capture(captureBuilder.build(),
                         captureCallback, null);
             }
           catch(CameraAccessException cae) {
           cae.printStackTrace();
          }
         }
```

This method prepares the request, setting some parameters and starts the capture session. As always, we use a `callback` method to be notified about the events.

9. Finally, we define a `callback` interface used by `AndroidCamera` to notify to the caller the most important events. The interface is defined as follows:

```
public static interface CameraListener {
       public void onCameraAvailable();
       public void onImageReady(ImageReader reader);
       }
```

The class that manages the camera in Android Things is ready and we can invoke it from the `MainActivity`.

Assembling the app

Finally, we can complete the Android Things app, modifying the `MainActivity`. The last step is handling the button that enables a user to take the picture:

1. Open `MainActivity.java` again and add the following lines in the `onCreate` method:

```
final AndroidCamera aCamera =
        new AndroidCamera(this, listener);
        aCamera.initCamera();
        aCamera.openCamera();
```

   In this way, we initialize the camera, setting the listener that will receive the event notification.

2. Moreover, in the same method, we reference the `ImageView` widget that will display the picture:

```
imgView = (ImageView) findViewById(R.id.img);
```

3. It is time to get the reference to the button used to take the picture:

```
btnPicture = (Button)
       findViewById(R.id.btnPicture);
       btnPicture.setEnabled(false);
```

   Initially, the button is disabled until the camera is ready to take the picture. For this purpose, the app uses the listener to know when the camera is ready (remember the `onCameraAvailable` method).

4. To handle the event triggered when the user clicks on the button we add the following lines:

```
btnPicture.setOnClickListener(
           new View.OnClickListener() {
        @Override
           public void onClick(View v) {
              Log.d(TAG, "Start caputring the image");
        aCamera.takePicture();
            }
        });
```

The method calls `takePicture` to capture the picture.

5. Finally, the app implements the `callback` interface that we have used previously in the constructor:

```
private AndroidCamera.CameraListener listener =
         new AndroidCamera.CameraListener() {
        @Override
           public void onCameraAvailable() {
                Log.d(TAG, "Camera Ready");
        btnPicture.setEnabled(true);
           }
       @Override
       public void onImageReady(ImageReader reader) {
          Log.d(TAG, "Image ready");
            Image img1 = reader.acquireLatestImage();
            ByteBuffer bBuffer =
```

```
            img1.getPlanes()[0].getBuffer();
        final byte[] buffer =
        new byte[bBuffer.remaining()];
            bBuffer.get(buffer);
            img1.close();
             runOnUiThread(new Runnable() {
        @Override public void run() {
        imgView.setImageBitmap(
               BitmapFactory.decodeByteArray(
               buffer, 0, buffer.length) );
             }
           });
         }};
```

In the first `callback` method, called `onCameraAvailable()`, the app enables the button as soon as the camera is ready. In the second method, called `onImageReady`, we update the view, setting the image in the `ImageView` widget. The code contained in this method is used to extract the image and to adapt it in a way that can be used in the `ImageView`.

Before running the app, we have to request the permission to use the camera. We do it in the `Manifest.xml` adding the following line:

```
<uses-permission android:name="android.permission.CAMERA" />
```

Congratulations! You have just built a spying system in Android Things that captures images. Now you can run the Android Things app and experiment with it.

Summary

In this chapter, we have built a system based on servo motors and cameras. We have learned how to control a servo motor using Android Things. Moreover, you gained knowledge about PWM and the role it plays in this context. Now you have all the necessary information to implement Android Things apps to control new kinds of peripherals. You are mastering several peripherals such as LEDs, sensors, buttons, camera, motors, and so on.

In the next chapter, we will cover how to integrate Android and Android Things and how to create an Android companion app that remotely controls an Android Things app.

Android with Android Things

In this chapter, we will cover how to integrate Android with Android Things. The aim of this chapter is to develop two Android apps that interact with Android Things. The convergence between mobile apps and an Internet of things application is an interesting field and we will describe the different strategies we can use to make these two ecosystems exchange data and information. In more detail, the chapter covers:

1. Different architecture we can use to integrate Android and Android Things.
2. How to develop an Android app that remotely controls a LED strip we have already built in Chapter 5, *Create a Smart System to Control Ambient Light*.
3. How to develop an Android app that shows data coming from sensors through Android Things. This app is a companion app for the Remote Weather station we built in Chapter 6, *Remote Weather Station*.

In this that chapter, we will re-use all the knowledge we have acquired during this book to build real-life Android and Android Things apps.

Architecture to connect Android and Android Things

In this paragraph, we will discover how we can integrate Android and Android Things. Nowadays, there are several market products that have a mobile companion app to interact with smart systems. Here are a few of them that we can remember:

1. Remote controlled smart light systems.
2. Alarm systems.
3. Remote controlled appliances.

Therefore, it is important to describe how we can integrate the smartphones ecosystems with Android Things. This section focuses its attention on Android smartphones but you can reuse the same strategies when integrating iOS apps with Android Things.

Generally speaking, there are three different scenarios:

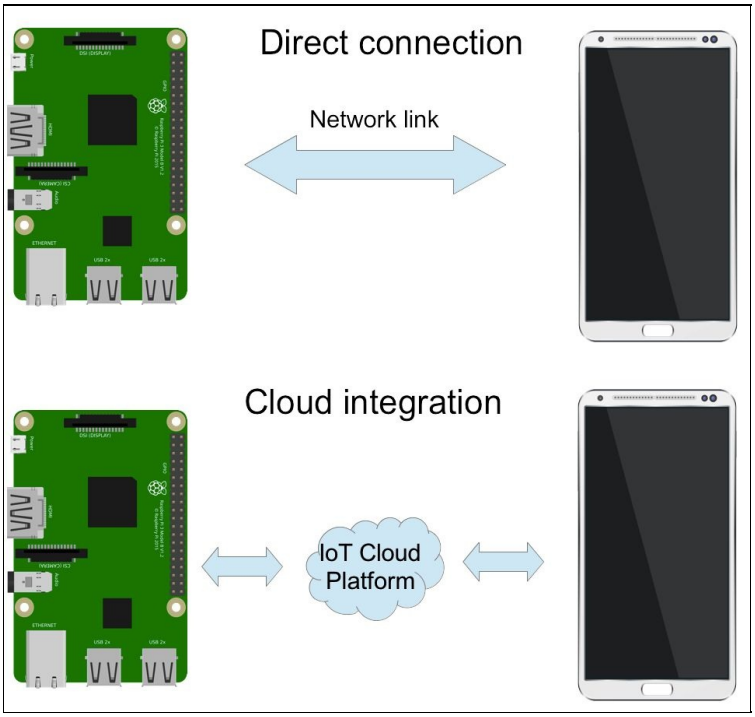
1. A smartphone controls a smart object (like the Android Things board) (*master- slave pattern*).
2. A smartphone receives data stream through the Android Things board.
3. A smartphone receives notifications from the Android Things system when an event occurs.

We covered the last point in Chapter 2, *Creating an Alarm System Using Android Things*, where we sent notifications to users' smartphones when the system detected motion in the detection area. Moreover, in Chapter 4, *Integrate Android Things with IoT Cloud Platforms*, we sent notifications using voice calls to the users' smartphone. Therefore, the last point in the integration architecture should be clear. We focus our attention on the first two strategies.

Generally, in the first two scenarios, the integration can happen in two different ways:

1. There is a direct link between the smartphone and the Android Things system.
2. Through cloud platforms.

The following picture visualizes these ways:



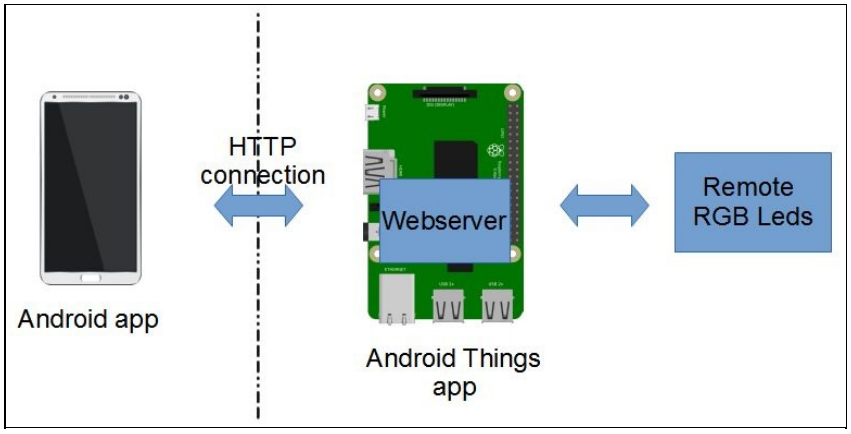
In the second scenario, the user's smartphone is connected to the Android Things board using an IoT cloud platform, or more generally, a cloud platform that provides some integration services like those we used to trigger a voice phone call.

In the direct integration scenario, there is a direct connection between the user's smartphone and the Android Things board. The connection can be established using several protocols like:

- WiFi
- Bluetooth
- Ethernet

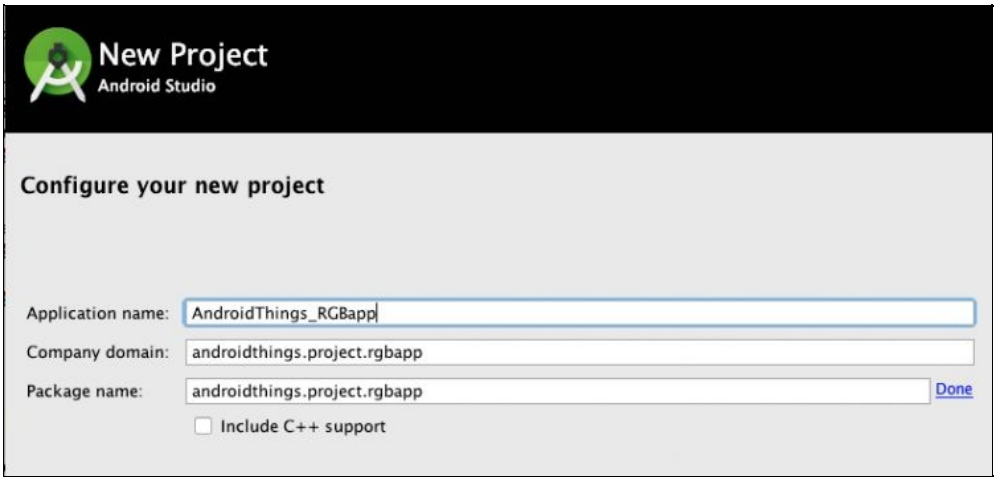
How to control a LED strip using an Android app

In this first Android project, we want to control an LED strip using an Android app. In more detail, the Android app connects **directly** to the Android Things app that, in turn, controls one or more LED strips through Arduino boards. As we stated in Chapter 5, *Create a Smart System to Control Ambient Light*, in this project the Android Things app behaves like a gateway acting as a unique point of access. This approach has several benefits, which were highlighted in Chapter 5, *Create a Smart System to Control Ambient Light*. In this application context, we exploit the built-in web server implemented in the Android Things app. The idea that stands behind this is to create an Android app that uses the HTTP protocol to remotely control the Android Things app (like a *master-slave* pattern). To this purpose, we can reuse the Android Things app we have developed and attach to it the Android app. In this way, we can control the Android Things app using a Web browser or the Android app that we will develop in this section. The following picture shows the project overview:

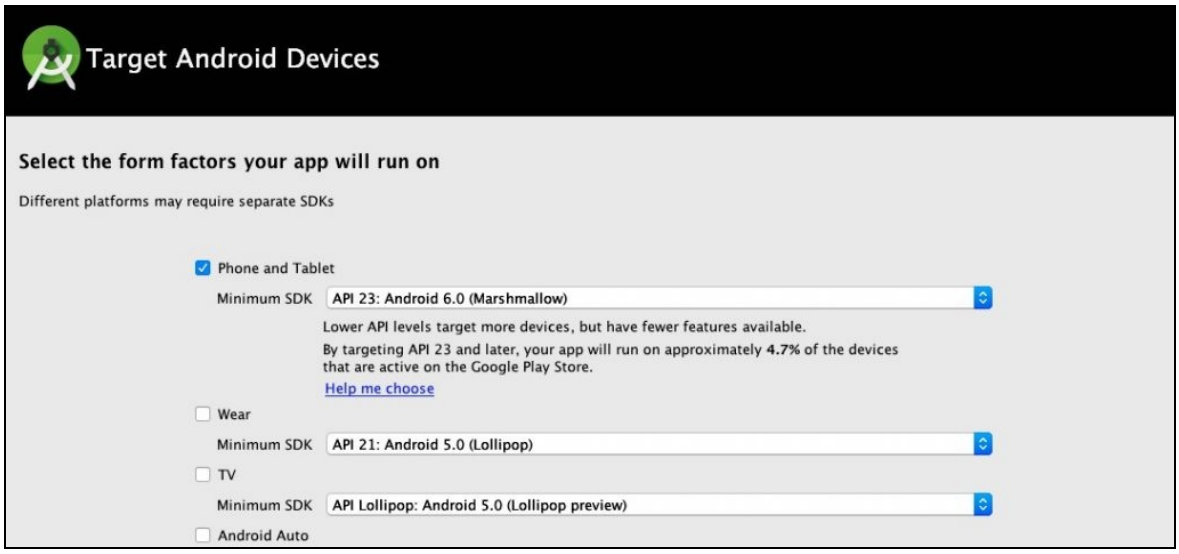


To do it, let's create a new **Android project**:

1. Open Android Studio and create a new project named `AndroidThing_RGBapp`:



2. Move to the next step, configuring Target Android Devices:



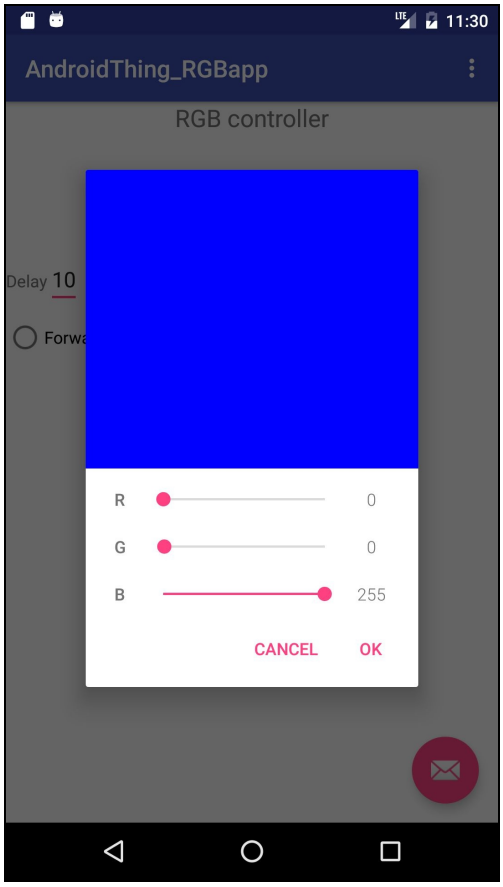
Minimum SDK: API level 23 (Android 6.0)

3. The last step is creating a **Basic Activity**:

That's all. Continue until you find the Finish button. Now you have created your Android project. As you will notice, it is very similar to the Android Things project we created in previous chapters. This app will be developed according to the *Material design* guidelines. If you are new to them, you should read this link (https://material.io/guidelines/). Moreover, this app uses a **Floating Action Button** (**FAB**), which is a button that represents the main action in this activity. In our app, the FAB represents the sending data action, which we will use to control the Android Things app. The following screenshot shows what the Android app UI looks like:

This chapter will not cover all of the app details because some steps are trivial. For example, the layout is very simple, and you can refer to the source code shipped with this book to know more. However, there are some interesting points that we will discuss in more detail because they represent the core concepts of this app.

The first important aspect is how the user selects the LED color. For this, we will use a simple dialog that opens when the user clicks the button. The result is shown as follows:



To implement it, follow these steps:

1. Open the `build.gradle` file (app level) and add the following lines into the `dependencies` tag:

```
compile 'me.priyesh:chroma:1.0.2'
```

2. Open `MainActivity.java` and add this piece of code:

```
    Button btn = (Button) findViewById(R.id.btnColor);
         btn.setOnClickListener(new View.OnClickListener()
          {
           @Override
           public void onClick(View v) {
           new ChromaDialog.Builder()
                           .initialColor(Color.BLUE)
                           .colorMode(ColorMode.RGB)
                           .onColorSelected(
                            new ColorSelectListener() {
            @Override public void onColorSelected(int color)
            {
             Log.d(TAG, "Color selected");
             red = Color.red(color);
             green = Color.green(color);
             blue = Color.blue(color);
                             }
                         });
                         .create()
                         .show( getSupportFragmentManager(),
                         "dialog");
               }
             });
```

   Here, `R.id.btnColor` is the ID of the `Button` widget used in the layout. The code is very simple: the app sets the color selection mode (*RGB*) and sets the listener to be informed when the user dismisses the dialog confirming the color selected.

   Moreover, the app uses the *red*, *green*, and *blue* components extracted from the color picked to control the RGB LEDs.

The management of the other widgets is very simple. It is useful to mention how to handle the group of radio buttons. In the UI, there are two groups:

1. The first group is used to handle the direction.
2. The second group is used to handle the type of operation we want to apply to the RGB LED strip.

The steps necessary to manage these two groups are the same, so we will only describe the second group of radios. To handle them, we have to:

1. Add the widget to the UI layout.
2. Add the methods to manage the widget in the Activity.

Regarding the first step, add the radio group widget to the app UI with the following lines:

```
<RadioGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintTop_toBottomOf="@id/dirGroup"
    android:layout_marginTop="20dp"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
```

```
    android:layout_marginBottom="20dp"
    app:layout_constraintBottom_toBottomOf="parent"
    android:orientation="horizontal">
<RadioButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/color"
    android:text="Set Color" android:onClick="onFunctionClick"/>
<RadioButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/clear"
    android:text="Clear"
    android:onClick="onFunctionClick"/>
<RadioButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/rainbow"
    android:text="Rainbow"
    android:onClick="onFunctionClick"/>
</RadioGroup>
```

Now we have to implement the method called when the user selects one of the radio buttons in the group. To do so, we have to add the following method to the activity:

```
public void onFunctionClick(View v) { switch (v.getId()) {
case R.id.color:
func = 0;
break;
case R.id.clear:
func = 1;
break;
case R.id.rainbow:
func = 2;
break;
}
}
```

This checks the ID of the widget so we know which one the user has selected. The other widgets are very simple, so we will not cover them.

Connecting the Android app to Android Things

Another interesting aspect is sending data to the Android Things app using HTTP. To this purpose, we will use a library, which we have already covered in previous chapters. This library is `OkHTTP`. As stated previously, we will send data when the user clicks on the *FAB*. The code to handle the HTTP connection is shown as follows:

```
fab.setOnClickListener(new View.OnClickListener() {
@Override
    public void onClick(View view) {
        String delVal = edt.getText().toString(); HttpUrl.Builder urlBuilder =
        HttpUrl.parse(baseUrl).newBuilder()
       .addQueryParameter("action",
                          String.valueOf(func))
       .addQueryParameter("red",
                          String.valueOf(red))
        .addQueryParameter("green",
                           String.valueOf(green))
        .addQueryParameter("blue",
                            String.valueOf(blue))
        .addQueryParameter("dir",
                            String.valueOf(direction))
        .addQueryParameter("delay", delVal);

        Request req = new Request.Builder()
                       .url(urlBuilder.build().toString())
                       .build();

       client.newCall(req).enqueue(new Callback() {
       @Override
            public void onFailure(Call call, IOException
          e)
           {
           Log.e(TAG, "Error");
           e.printStackTrace();
           }
       @Override
           public void onResponse(Call call,
                       Response response) throws
         IOException {
                  Log.i("TAG", "Response.." +
                  response.body().string());
           }
        });
      }
})
```

The code is quite simple:

1. The app sets the listener to be informed when the user clicks on the FAB.
2. The app prepares the URL adding the parameters, such as red, green, blue values, and so on.
3. The app invokes the URL exposed by the Android Things app passing the parameters (`GET` request).
4. The app sets a listener to be informed when the Android Things app, through the web server, sends back the response.

Before running the app, do not forget to add the permission to use internet in your `Manifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Now you can run the app using an Android emulator or your smartphone. If you want to test the app, you have to follow these steps:

1. Connect your Arduino board to the RGB LED strip, as described in Chapter 5, *Create a Smart System to Control Ambient Light*.
2. Install in your Android Things board the app that handles the RGB LEDs. If you use Intel Edison with an Arduino breakout kit, you do not have to modify the Android Things app source code because the app runs the web server by default. If you use Raspberry Pi 3, you have to enable the web server because the app does not run it by default.
3. Get the Android Things board's IP and replace the IP used in Android app with the correct one.

Now you can test the Android Things app.

Congratulations!! You have built a system that integrates Android and Android Things using a direct connection based on a master-slave pattern.

How to develop an Android app that retrieves data from Android Things

In this section, we will cover another integration scenario where an Android app retrieves data from Android Things. While in the previous scenario we used the Android app to control Android Things, here, in this context, we want to retrieve information from sensors connected to the Android Things board. To demonstrate how to do this, we will reuse the project we developed in Chapter 6, *Remote Weather Station*. To retrieve the data, we have several options, but we are interested in these two following strategies:

1. Using the MQTT protocol.
2. Connecting the Android app to Android Things using Bluetooth.

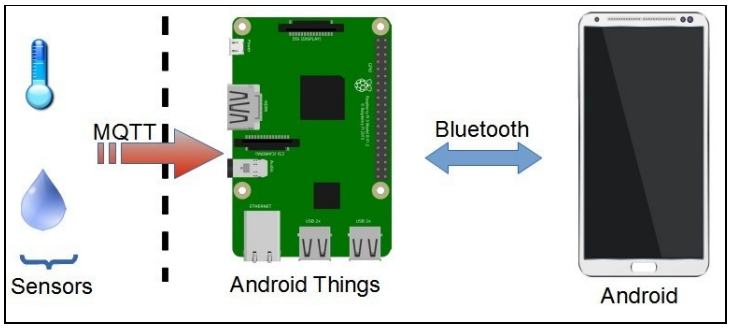
In the first option, in order to retrieve data from sensors, we could implement an Android app that uses MQTT through the Android Things app. As you will remember, in the Remote Weather station project we used MQTT to connect several boards and applications. In this context, we simply have to implement an Android MQTT subscriber app. If we do not want to develop it, there are several Android apps available on the Google play store. We can download one of them and connect it to the MQTT broker. As soon as we have correctly configured the app, we start receiving the data. This is the simplest way to integrate the existing Android Things app with Android. There is also another method we can exploit to make Android and the Android Thing app exchange data. This method uses the Bluetooth connection. As you may already know, Bluetooth is an industrial standard widely used to exchange data in **Wireless Personal Area Network** (**WPAN**). It provides an efficient way of exchanging information between devices over a short range. Both Android and Android Things support a Bluetooth connection. More generally, the concepts you learn here can be applied to other projects where there is the need to exchange data. Therefore, it is important you understand how to use Bluetooth in Android and Android Things.

How to implement a Bluetooth connection

In order to connect the Android app to Android Things through a Bluetooth connection, we have to follow the following steps:

1. Create an Android app that behaves as a client connecting to the Android Things app that plays the server role.
2. Modify the Android Things app, previously implemented, by adding the Bluetooth feature.
3. Modify the Android Things app to send data through Bluetooth as it receives data from MQTT.

The following figure shows the project overview:



You can use this architecture with other kinds of projects. For example, you can apply this architecture to the project developed in Chapter 3, How to Make an Environmental Monitoring System, where sensors are connected directly to Android Things board.

The idea that stands behind this project is to create a server that accepts a connection from clients through a Bluetooth connection. In order to make it work, we have to implement:

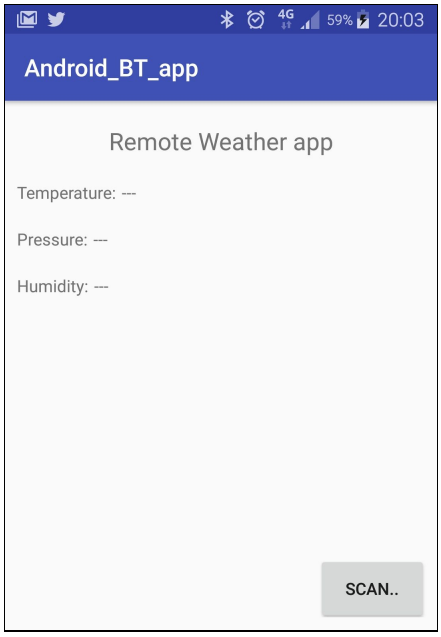
1. The client, that is, the Android app.
2. The server, that is, the Android Things app.

Let us see how to implement each step described previously.

Creating the Android app

In this step, as stated before, we have to create an Android app that connects to the Android Things board using a Bluetooth connection. There are several ways we can implement the app. Firstly, we could exploit the Android Bluetooth API (https://developer.android.com/guide/topics/connectivity/bluetooth.html) or we can use an open source library that does the heavy work. The use of the Android Bluetooth API is out of the scope of this book, as we are interested in understanding how Android and Android Things can exchange data. For this purpose, then, we will use a simple Android Bluetooth library that helps us to develop the client and server side easily.

There are several open source libraries available; in this project, we will use `SimpleBluetoothLibrary` (https://github.com/DeveloperPaul123/SimpleBluetoothLibrary). The Android app UI is shown as follows:



To develop the Android app, follow the following steps:

1. Create a new Android Project in Android Studio, as described in the previous paragraph.
2. In `build.gradle` at the project level, add the following line in `allprojects` tag:

```
maven {url "https://jitpack.io"}
```

3. In `build.gradle` at the app level we have to declare the dependency to the Bluetooth library. Therefore, add this line in `dependencies` tag:

```
            compile 'com.github.DeveloperPaul123:SimpleBluetoothLibrary:1.5.1'
```

4. Now we are ready to develop the app. We will focus on the Bluetooth aspects and how to implement them without covering the UI aspects because they are very simple. In `MainActivity.java`, we have to add the method to initialize Bluetooth:

```
private void initBT() {
        btConnection = new SimpleBluetooth(this, this);
        btConnection.setSimpleBluetoothListener(
           new SimpleBluetoothListener() {
        @Override
```

```
        public void onBluetoothDataReceived(
            byte[] bytes, String data) {
      super.onBluetoothDataReceived(
                bytes, data);
            Log.d(TAG, "Data received");
            //Update the UI
            }
      @Override
        public void onDeviceConnected(
           BluetoothDevice device) {
      super.onDeviceConnected(device);
           Log.d(TAG, "Device connected"
      + device.getName());
            }
      @Override
           public void
               onDeviceDisconnected(
           BluetoothDevice device) {
      super.onDeviceDisconnected(device);
           Log.d(TAG, "Device disconnected"
                       + device.getName());
            }
      @Override
           public void onDiscoveryStarted() {
      super.onDiscoveryStarted();
           Log.d(TAG, "Discovery started");
           }
      @Override
           public void onDiscoveryFinished() {
      super.onDiscoveryFinished();
           Log.d(TAG, "Discovery finished");}
      @Override
            public void onDevicePaired(
                  BluetoothDevice device {
      super.onDevicePaired(device);
             Log.d(TAG, "Device paiered" +
                    device.getName());
             }
      @Override
          public void onDeviceUnpaired(
             BluetoothDevice device) {
      super.onDeviceUnpaired(device);
          Log.d(TAG, "Device unpaired"
            + device.getName());
        }
      });
      btConnection.initializeSimpleBluetooth();
      btConnection.setInputStreamType(
        BluetoothUtility.InputStr eamType.NO RMAL);
     }
```

The code is quite simple. The app initializes the Bluetooth library (`SimpleBluetooth`) in the first line. In the next step, it declares a listener to be informed about Bluetooth events. As you will notice, there are several events included.

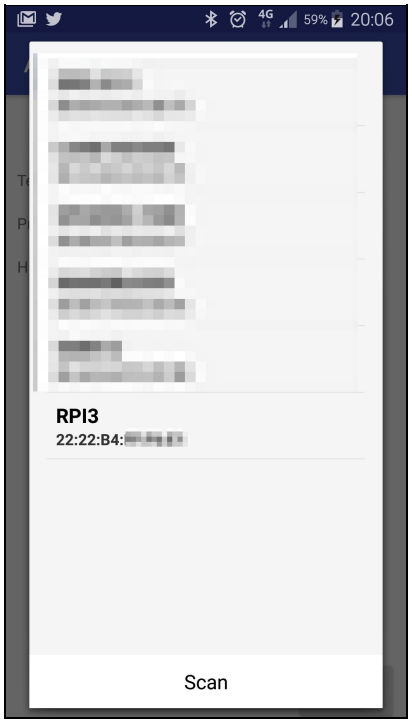
Considering we are implementing a client, we are interested in retrieving messages from the server. For this reason, we override `onBluetoothDataReceived`. In this method, we will update the app UI according to the data received. At the end of the method, we start Bluetooth and we set the data stream type.

5. In the `onCreate` method we have to invoke the `initBT()` method.
6. Moreover, it is necessary that the app has a button that can be used to scan other Bluetooth devices nearby (like the Android Things app). In this way, the Android app can connect to the Android Things board. For this reason, the app UI has a button called scan. When the user clicks this button, we use the library to detect devices nearby:

```
Button scanBtn = (Button)
```

```
                findViewById(R.id.scan_button);
    scanBtn.setOnClickListener(
       new View.OnClickListener() {
    @Override
          public void onClick(View v) {
    btConnection.scan(SCAN_REQUEST);
          }
  });
```

When the app invokes the scan method, the library starts another activity that will return to `MainActivity` the information of the device selected by the user. This activity is provided by the `SimpleBluetooth` library so that we do not have to write a single line of code. Notice how fast developing the app is in this way. The app UI during the scan is shown as follows:



7. In this step, we have to override the method `onActivityResult` that is called when the activity started at *step 6* returns the information to the caller:

```
@Override
        protected void onActivityResult(
           int requestCode, int resultCode, Intent data) {
        super.onActivityResult(
           requestCode, resultCode, data);

      if (requestCode == SCAN_REQUEST) {
      if (resultCode == RESULT_OK) {
      serverMacAdd = data.getStringExtra(
      DeviceDialog.DEVICE_DIALOG_DEVICE_ADD RESS_EXTRA
      );
      Log.d(TAG, "Device Add ["+serverMacAdd+"]");
     btConnection.connectToBluetoothServer(serverMacAdd);
              }
          }
      }
```

The information returned by the scan activity is the *device* that represents the device selected by the user during the scanning process. In more detail, we are interested in the mac address of the device. The mac address is the identification of the device we have selected. In this context, this mac address represents the Android Things board. We use this address to connect to the server through the Bluetooth connection.

That's all! In a few steps, we have implemented a simple Android app that behaves as a client and we will now use it to connect to the Android Things app that behaves like a server. Once the connection is established these apps will exchange data. Before running the app, we have to modify `Manifest.xml`, adding the following permission to use Bluetooth:

```
<uses-permission
    android:name="android.permission.BLUETOOTH" />
<uses-permission
    android:name="android.permission.BLUETOOTH_ADMIN" />
```

The client side is ready! Let's now focus on the server side.

initBT();

private void initBT() {
 Log.d(TAG, "BT init...");
 btConnection = new SimpleBluetooth(this, this);
 btConnection.setSimpleBluetoothListener(.....);
 btConnection.makeDiscoverable(600);
 btConnection.initializeSimpleBluetooth();
 btConnection.setInputStreamType(
 BluetoothUtility.InputStreamType.NORMAL);
 btConnection.createBluetoothServerConnection();
 }

btConnection.sendData(payload);

<uses-permission
 android:name="android.permission.BLUETOOTH" />
<uses-permission
 android:name="android.permission.BLUETOOTH_ADMIN" />

That's all the integration is now complete. Now you can run the app and check it.

Summary

Aw we complete this chapter, you have learned how to integrate Android and Android Things. We have explored the different architectures we can use according to the scenario where the apps will work. The knowledge you gained in this chapter can be reused in different scenarios. During this book, we have explored several aspects of Android Things, and different ways we can exploit the power of Android Things OS, integrating with IoT cloud platforms or with other IoT development boards.