

Gastón C. Hillar

MQTT Essentials - A Lightweight IoT Protocol

The preferred IoT publish-subscribe lightweight
messaging protocol



Packt

MQTT Essentials - A Lightweight IoT Protocol

Table of Contents

[MQTT Essentials - A Lightweight IoT Protocol](#)

[Credits](#)

[About the Author](#)

[Acknowledgment](#)

[About the Reviewer](#)

[www.PacktPub.com](#)

[Why subscribe?](#)

[Customer Feedback](#)

[Dedication](#)

[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. Lightweight Messaging with MQTT 3.1.1 and Mosquitto](#)

[Understanding convenient scenarios for the MQTT protocol](#)

[Working with the publish-subscribe pattern](#)

[Working with message filtering](#)

[Understanding the MQTT puzzle -- clients, servers, and connections](#)

[Installing a Mosquitto broker on Ubuntu Linux](#)

[Installing a Mosquitto broker on macOS](#)

[Installing a Mosquitto broker on Windows](#)

[Subscribing to topics](#)

[Publishing messages](#)

[Unsubscribing from topics](#)

[Working with best practices when creating topics](#)

[Understanding wildcards](#)

[Learning about the different Quality of Service levels](#)

[Working with at least once delivery](#)

[Working with exactly once delivery](#)

[Test your knowledge](#)

[Summary](#)

[2. Securing an MQTT Mosquitto Server](#)

[Securing a Mosquitto server](#)

[Generating a private certificate authority to use TLS with Mosquitto](#)

[Creating a certificate for the Mosquitto server](#)

[Configuring TLS transport security in Mosquitto](#)

[Testing the MQTT TLS configuration with MQTT.fx](#)

[Testing the MQTT TLS configuration with MQTT-spy](#)

[Creating a certificate for each MQTT client](#)

[Configuring TLS client certificate authentication in Mosquitto](#)

[Testing the MQTT TLS client authentication with MQTT.fx](#)

[Testing the MQTT TLS configuration with MQTT-spy](#)

[Forcing the TLS protocol version](#)

[Test your knowledge](#)

[Summary](#)

[3. Controlling a Drone with Python](#)

[Understanding the requirements to control a drone with MQTT](#)

[Defining the topics and commands](#)

[Installing paho-mqtt for Python](#)

[Connecting the client to the MQTT server with Python](#)

[Understanding callbacks](#)

[Subscribing to topics with Python](#)

[Creating a class to represent a drone](#)

[Receiving messages with Python](#)

[Processing commands with Python](#)

[Sending messages with Python](#)

[Working with the network loop with Python](#)

[Test your knowledge](#)

[Summary](#)

[4. Interacting with Sensors in Java](#)

[Understanding the requirements](#)

[Defining the topics and commands](#)

[Installing the Eclipse Paho Java Client](#)

[Creating a security helper class to establish a TLS secured connection](#)

[Configuring a TLS secured connection in Java](#)

[Working with asynchronous notifications and callbacks in Java](#)

[Subscribing to topics and understanding notification events in Java](#)

[Processing commands that interact with sensors in Java](#)

[Working with last will and testament in Java](#)

[Publishing messages with commands to target sensors](#)

[Test your knowledge](#)

[Summary](#)

[5. Controlling Home Automation in JavaScript](#)

[Understanding the requirements](#)

[Defining the topics and commands to control home automation](#)

[Enabling WebSockets for a Mosquitto server](#)

[Testing WebSockets with the Eclipse Paho client JavaScript utility](#)

[Understanding the Eclipse Paho JavaScript asynchronous client](#)

[Creating an HTML 5 web page to control LEDs](#)

[Publishing messages to target remote devices in JavaScript](#)

[Processing the received messages in JavaScript](#)

[Working with WebSockets to connect to the MQTT server](#)

[Subscribing to topics in JavaScript](#)

[Processing commands that interact with LEDs in Node.js](#)

[Running the home automation web application](#)

[Debugging asynchronous code with Chrome Developer Tools \(CDT\) in JavaScript](#)

[Securing MQTT over WebSockets with TLS](#)

[Test your knowledge](#)

[Summary](#)

[6. Interacting with Actuators in Swift](#)

[Understanding the requirements](#)

[Defining topics and commands](#)

[Enabling connections without TLS for a Mosquitto server](#)

[Installing the CocoaMQTT client library to create a new iOS app](#)

[Building the UI for the iOS app that will control an actuator](#)

[Understanding the CocoaMQTT client in Swift 3](#)

[Connecting to the MQTT server and subscribing to topics in Swift](#)

[Publishing messages to target actuators in Swift](#)

[Processing commands that interact with actuators in Node.js](#)

[Running the iOS app that controls an actuator with MQTT](#)

[Debugging events in Swift](#)

[Test your knowledge](#)

[Summary](#)

[A. Exersise Answers](#)

[Chapter 1, Lightweight Messaging with MQTT 3.1.1 and Mosquitto](#)

[Chapter 2, Securing an MQTT Mosquitto Server](#)

[Chapter 3, Controlling a Drone with Python](#)

[Chapter 4, Interacting with Sensors in Java](#)

[Chapter 5, Controlling Home Automation in JavaScript](#)

[Chapter 6, Interacting with Actuators in Swift](#)

MQTT Essentials - A Lightweight IoT Protocol

MQTT Essentials - A Lightweight IoT Protocol

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: April 2017

Production reference: 1050417

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78728-781-5

www.packtpub.com

Credits

Author	Copy Editor
Gastón C. Hillar	Muktikant Garimella
Reviewer	Project Coordinator
Ben Howes	Ulhas Kambali
Commissioning Editor	Proofreader
Smeet Thakkar	Safis Editing
Acquisition Editor	Indexer
Reshma Raman	Aishwarya Gangawane
Content Development Editor	Graphics
Vikas Tiwari	Abhinash Sahu
Technical Editor	Production Coordinator
Jijo Maliyekal	Arvindkumar Gupta

About the Author

Gastón C. Hillar is Italian and has been working with computers since he was eight. He began programming with the legendary Texas TI-99/4A and Commodore 64 home computers in the early 80s. He has a bachelor's degree in computer science (graduated with honors), and an MBA (graduated with an outstanding thesis). At present, Gastón is an independent IT consultant and freelance author who is always looking for new adventures around the world.

He has been a senior contributing editor at Dr. Dobb's and has written more than a hundred articles on software development topics. Gastón was also a former Microsoft MVP in technical computing. He has received the prestigious Intel® Black Belt Software Developer award eight times.

He is a guest blogger at Intel® Software Network (<http://software.intel.com>). You can reach him at gastonhillar@hotmail.com and follow him on Twitter at <http://twitter.com/gastonhillar>. Gastón's blog is <http://csharpmulticore.blogspot.com>.

He lives with his wife, Vanesa, and his two sons, Kevin and Brandon.

Acknowledgment

At the time of writing this book, I was fortunate to work with an excellent team at Packt Publishing, whose contributions vastly improved the presentation of this book. Reshma Raman allowed me to provide her ideas for a book dedicated to MQTT, and I jumped into the exciting project of teaching the features of one of the most important protocols for IoT projects with examples in many different programming languages. Vikas Tiwari helped me realize my vision for this book and provided many sensible suggestions regarding the text, the format, and the flow. The reader will notice his great work. It's been great working with Reshma and Vikas in another project and I can't wait to work with them again. I would like to thank my technical reviewers and proofreaders for their thorough reviews and insightful comments. I was able to incorporate some of the knowledge and wisdom they have gained in their many years in the software development industry. This book was possible because they gave valuable feedback.

The entire process of writing a book requires a huge number of lonely hours. I wouldn't be able to write an entire book without dedicating some time to play soccer against my sons, Kevin and Brandon, and my nephew, Nicolas. Of course, I never won a match. However, I did score a few goals.

About the Reviewer

Ben Howes is the founder and lead consultant at Zoetrope Ltd, a specialist IoT consultancy and product development firm in the UK. Ben has been creating connected hardware for over 10 years and has worked across projects spanning from start-ups to multinational deployments.

I'd like to thank Richard Webb, my co-founder, for starting Zoetrope with me.

www.PacktPub.com

For support files and downloads related to your book, please visit 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 at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

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



<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/1787287815>.

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!

Dedication

To my sons, Kevin and Brandon, and my wife, Vanesa

Preface

MQTT is the preferred IoT publish-subscribe lightweight messaging protocol. If you develop IoT, web applications, mobile apps, or a combination of these solutions, you must learn how MQTT and its lightweight messaging system work. Of course, it is extremely important to take security into account when working with this protocol.

Most of the times, when you work with IoT solutions, you won't be using a single programming language. Instead, you will use a combination of different programming languages and code running on different IoT boards that might use diverse operating systems. MQTT has its own specific vocabulary and different working modes. Learning MQTT is challenging because it includes too many abstract concepts that require real-life examples to be easy to understand.

This book will allow you to dive deep into the latest version of the MQTT protocol: 3.1.1. You will learn to work with Mosquitto MQTT server, command-line tools, and GUI tools to allow you to understand how everything works with MQTT and the possibilities that this protocol provides for your projects. You will learn security best practices and you will use them for a Mosquitto MQTT server.

This book will provide you with examples in the following programming languages, which are extremely popular in IoT, web applications, and mobile apps: Python, Java, HTML 5, JavaScript, Node.js, and Swift 3. You will control a drone, interact with sensors, control home automation, and interact with actuators by exchanging MQTT messages with the different programming languages and client libraries.

You will be able to run the examples on a wide range of modern IoT boards, such as Raspberry Pi 3, Raspberry Pi 2 Model B, Raspberry Pi Zero, Intel Joule, Intel Edison, and Intel Galileo Gen 2. However, any other board that supports the programming languages explained in the book will be able to run the samples.

What this book covers

[Chapter 1](#), *Lightweight Messaging with MQTT 3.1.1 and Mosquitto*, starts our journey towards understanding the MQTT protocol. You will learn convenient scenarios for this protocol, the details of the publish-subscribe pattern, and message filtering. You will learn basic concepts related to MQTT and understand the different components: clients, brokers or servers, and connections. You will learn to install a Mosquitto broker or server on Linux, macOS, and Windows. You will learn best practices related to topics and single-level and multilevel wildcards. We will study in detail the different Quality of Service levels supported by MQTT and when it is convenient to use each of them. We will analyze their advantages and disadvantages.

[Chapter 2](#), *Securing an MQTT Mosquitto Server*, covers generating a private certificate authority, a server certificate, and client certificates to enable TLS transport security and TLS client authentication with Mosquitto. Communications between the MQTT clients and the MQTT server will be encrypted. We will work with OpenSSL to generate the digital certificates. We will test the MQTT TLS configuration with MQTT.fx, MQTT-spy, and the Mosquitto command-line utilities. We will force Mosquitto to use only a specific TLS version. This way, we will understand how to encrypt communications between the MQTT clients and the Mosquitto MQTT server.

[Chapter 3](#), *Controlling a Drone with Python*, analyzes the requirements for controlling a drone with MQTT messages. We will define the topics that we will use and the commands that will be part of the messages' payloads to control a drone. Then, we will work with the Paho Python Client to write Python code that connects an MQTT client to the MQTT server. We will understand the methods we need to call for the Paho Python Client and their parameters. We will analyze how callbacks work and we will be able to publish messages to topics, subscribe to topic filters, and receive and process messages. We will write code that processes commands for a drone with Python. The code will be able to run on different IoT platforms, including Intel Joule boards and Raspberry Pi 3, and also on any computer that is capable of executing Python 2.7.x or Python 3.x. We will work with the network loop for the MQTT client in Python. We will work with a GUI utility, command-line-based utilities, and, finally, with Python code to control the drone and display the feedback provided by the drone.

[Chapter 4](#), *Interacting with Sensors in Java*, analyzes the requirements for controlling sensors wired to different IoT boards with MQTT messages and Java code. We will define the topics that we will use and the commands that will be part of the messages' payloads to control sensors. We will take advantage of the last will and testament feature of MQTT with Java. We will work with the Eclipse Paho Java Client to write Java code that will connect a non-blocking (asynchronous) MQTT client to the MQTT server. We will create a security helper class to establish a TLS secured connection, and we will work with asynchronous notifications and callbacks in Java. We will subscribe to topics, understand how notification events work, and we will write code that processes commands that interact with sensors. Finally, we will work with a GUI utility to interact with our Java application through MQTT messages. We will be able to turn on and turn off the sensors and retrieve data from them.

[Chapter 5](#), *Controlling Home Automation in JavaScript*, analyzes the requirements to control LEDs wired to different IoT boards with MQTT messages over WebSockets. We will define the topics that we will use and the commands that will be part of the message's payloads to set the colors for LEDs. Then,

we will create an HTML 5 web page combined with JavaScript and the Eclipse Paho JavaScript asynchronous client to control LEDs. We will write code to publish messages to target remote devices in JavaScript and we will be able to process received messages to provide feedback to the user. We will work with MQTT.js to write a Node.js script to process commands that simulate the interaction with LEDs on different IoT boards. We will be able to run the home automation web application and understand how all the pieces work together by exchanging MQTT over WebSockets messages. Finally, we will understand the necessary steps to work with MQTT over WebSockets with TLS.

[Chapter 6](#), *Interacting with Actuators in Swift*, analyzes the requirements for controlling actuators wired to different IoT boards with MQTT messages. We will define the topics that we will use and the commands that will be part of the message's payloads to turn on and turn off motors. Then, we will create an iOS app with Xcode and Swift 3, combined with the CocoaMQTT client, to control a motor. We will write code to publish messages to target remote devices in Swift 3 and we will be able to process received messages to provide feedback to the user. We will work with MQTT.js to write a Node.js script to process commands that simulate the interaction with motors on the different IoT boards. We will be able to run the iOS app and the Node.js script and understand how all the pieces work together by exchanging MQTT messages.

What you need for this book

You will need any computer or device capable of running Python 3.x or greater, Chrome 56.x or greater, JDK 8 or greater, and Node.js 6.7.x or greater.

In order to work with Mosquitto 1.4.11 or greater, you will need a computer with any of the operating systems supported by this version of Mosquitto. The list is available at <https://mosquitto.org/download/>.

In order to work with Xcode 8.x, you will need a Mac computer capable of running OS X 10.11.5 or later, with 8 GB RAM.

Who this book is for

This book is a great resource for developers who want to learn more about the MQTT protocol to apply it to their individual IoT projects. Prior knowledge of working with IoT devices is essential.

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: "If a client specifies an empty value as the `clientId`."

A block of code is set as follows:

```
# Replace localhost with the IP for the Mosquitto
# or other MQTT server
mqtt_server_host = "localhost"
mqtt_server_port = 8883
mqtt_keepalive = 60
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<div id="led-2">
<span>
Select color for LED #2
</span>
```

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

```
sudo apt-get update
```

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 the text like this: "In order to download new modules, we will go to **Files | Settings | Project Name | Project Interpreter**."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for 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/MQTT-Essentials-A-Lightweight-IoT-Protocol>. 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/MQTTEssentialsALightweightIoTProtocol_Colo

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.

Chapter 1. Lightweight Messaging with MQTT

3.1.1 and Mosquitto

In this chapter, we will start our journey towards the usage of the preferred IoT publish-subscribe lightweight messaging protocol in diverse IoT solutions combined with mobile apps and web applications. We will learn how MQTT and its lightweight messaging system work. We will learn MQTT basics, the specific vocabulary for MQTT, and its working modes. We will use different utilities and diagrams to understand the most important concepts related to MQTT. We will:

- Understand convenient scenarios for the MQTT protocol
- Work with the publish-subscribe pattern
- Work with message filtering
- Learn basic concepts related to MQTT
- Understand the MQTT puzzle: clients, servers or brokers, and connections
- Install a Mosquitto MQTT server on Linux, macOS, or Windows
- Publish messages
- Subscribe to topics
- Unsubscribe from topics
- Work with best practices when creating topics
- Understand wildcards
- Learn about the different Quality of Service levels
- Work with at least once delivery
- Work with exactly once delivery

Understanding convenient scenarios for the MQTT protocol

Imagine that we have dozens of different devices that must exchange data between them. These devices have to request data to other devices and the devices that receive the requests must respond with the demanded data. The devices that requested the data must process the data received from the devices that responded with the demanded data.

The devices are **Internet of Things (IoT)** boards that have dozens of sensors wired to them. We have the following IoT boards with different processing powers: Raspberry Pi 3, Raspberry Pi Model B, Intel Edison, and Intel Joule 570x. Each of these boards has to be able to send and receive data. In addition, we want many mobile devices to be able to send and receive data, some of them running iOS and others Android. We have to work with many programming languages.

We want to send and receive data in near real time through the Internet and we might face some network problems, that is, our wireless networks are somewhat unreliable and we have some high-latency environments. Some devices have low power, many of them are powered by batteries and their resources are scarce. In addition, we must be careful with the network bandwidth usage because some of the devices use metered connections.

Tip

A metered connection is a network connection in which we have a limited amount of data usage per month. If we go over this amount of data, we get billed extra charges.

We can use HTTP requests and build a publish-subscribe model to exchange data between different devices. However, there is a protocol that has been specifically designed to be lighter than the HTTP 1.1 protocol and work better when unreliable networks are involved and connectivity is intermittent. The **MQ Telemetry Transport (MQTT)** is better suited for this scenario in which many devices have to exchange data between themselves in near real time through the Internet and we need to consume the least possible network bandwidth.

The MQTT protocol is a **Machine-to-Machine (M2M)** and IoT connectivity protocol. MQTT is a lightweight messaging protocol that works with a broker-based publish-subscribe mechanism and runs on top of **Transmission Control Protocol / Internet Protocol (TCP/IP)**.

The following diagram shows the MQTT protocol on top of the TCP/IP stack:

MQTT

TCP

IP

Note

The most popular versions of MQTT are 3.1 and 3.1.1. In this book, we will work with MQTT 3.1.1. Whenever we reference MQTT, we are talking about MQTT 3.1.1, that is, the newest version of the protocol. The MQTT 3.1.1 specification has been standardised by the OASIS consortium. In addition, MQTT 3.1.1 became an ISO standard (ISO/IEC 20922) in 2016.

MQTT is lighter than the HTTP 1.1 protocol, and therefore, it is a very interesting option whenever we have to send and receive data in near real time with a publish-subscribe model and we require the lowest possible footprint. MQTT is very popular in IoT, M2M, and embedded projects, but it is also gaining presence in web applications and mobile apps that require assured messaging and an efficient message distribution. As a summary, MQTT is suitable for the following application domains in which data exchange is required:

- Asset tracking and management
- Automotive telematics
- Chemical detection
- Environment and traffic monitoring
- Field force automation

- Fire and gas testing
- Home automation
- **In-Vehicle Infotainment (IVI)**
- Medical
- Messaging
- **Point of Sale (POS) kiosks**
- Railway
- **Radio-Frequency Identification (RFID)**
- **Supervisory Control and Data Acquisition (SCADA)**
- Slot machines

As a summary, MQTT was designed to be suitable to support the following typical challenges in IoT, M2M, embedded, and mobile applications:

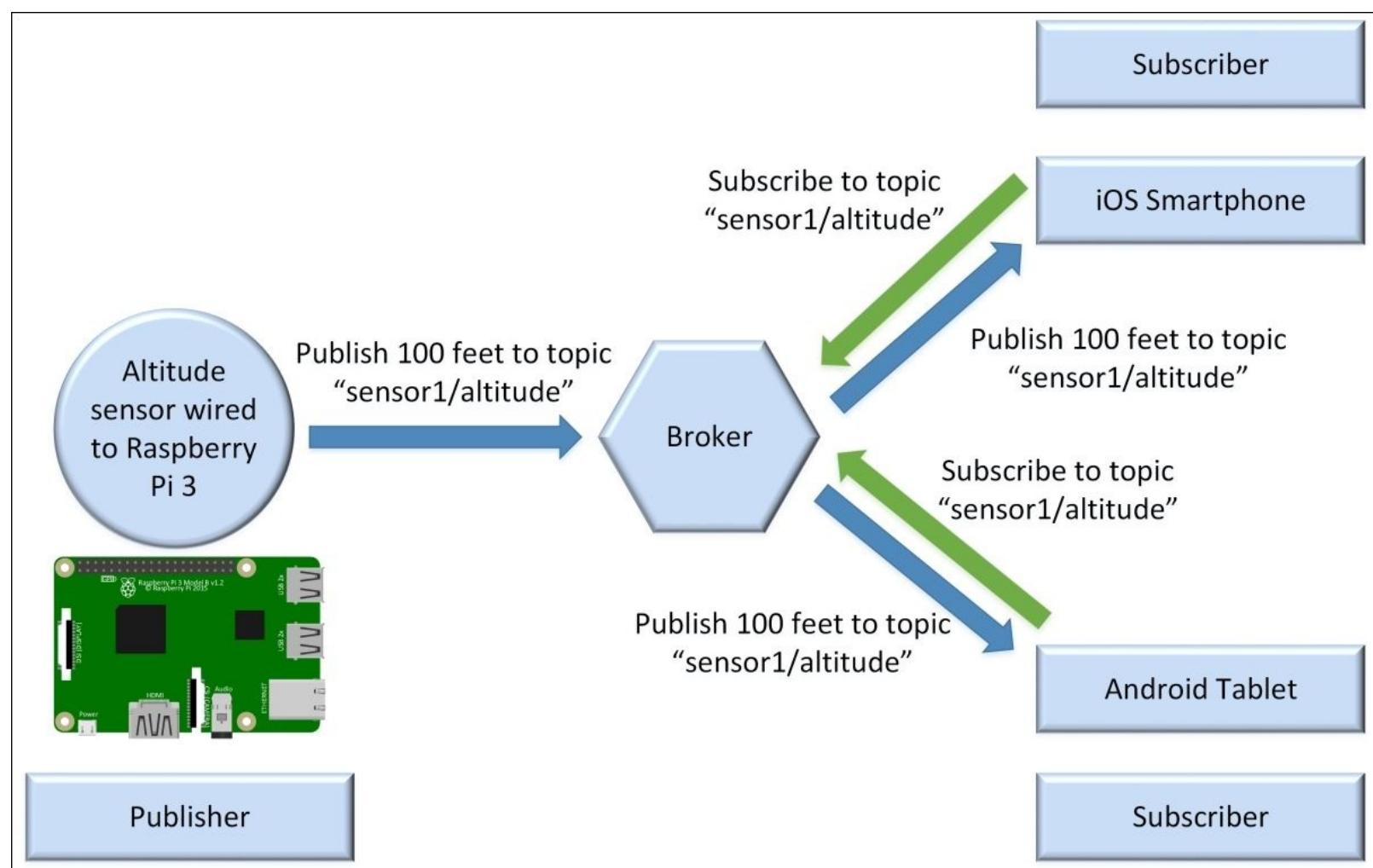
- Be lightweight to make it possible to transmit high volumes of data without huge overheads
- Distribute minimal packets of data in huge volumes
- Support an event-oriented paradigm with asynchronous bidirectional low latency push delivery of messages
- Easily emit data from one client to many clients
- Make it possible to listen for events whenever they happen (event-oriented architecture)
- Support always-connected and sometimes-connected models
- Publish information over unreliable networks and provide reliable deliveries over fragile connections
- Work very well with battery-powered devices or require low power consumption
- Provide responsiveness to make it possible to achieve near real-time delivery of information
- Offer security and privacy for all the data
- Be able to provide the necessary scalability to distribute data to hundreds of thousands of clients

Working with the publish-subscribe pattern

Before we dive deep into MQTT, we must understand the publish-subscribe pattern, also known as the pub-sub pattern. In the publish-subscribe pattern, a client that publishes a message is decoupled from the other client or clients that receive the message. The clients don't know about the existence of the other clients. A client can publish messages of a specific type and only the clients that are interested in specific types of messages will receive the published messages.

The publish-subscribe pattern requires a **broker**, also known as **server**. All the clients establish a connection with the broker. The client that sends a message through the broker is known as the **publisher**. The broker filters the incoming messages and distributes them to the clients that are interested in the type of received messages. The clients that register to the broker as interested in specific types of messages are known as **subscribers**. Hence, both publishers and subscribers establish a connection with the broker.

It is easy to understand how things work with a simple diagram. The following diagram shows one publisher and two subscribers connected to a broker:



A Raspberry Pi 3 board with an altitude sensor wired to it is a publisher that establishes a connection with the broker. An iOS smartphone and an Android tablet are two subscribers that establish a connection with the broker.

The iOS smartphone indicates to the broker that it wants to subscribe to all the messages that belong to the **sensor1/altitude** topic. The Android tablet indicates the same to the broker. Hence, both the iOS smartphone and the Android tablet are subscribed to the **sensor1/altitude** topic.

Tip

A **topic** is a named logical channel and it is also referred to as a channel or subject. The broker will send publishers only the messages published to topics to which they are subscribed.

The Raspberry Pi 3 board publishes a message with **100 feet** as the payload and **sensor1/altitude** as the topic. The board, that is, the publisher, sends the publish request to the broker.

Tip

The data for a message is known as **payload**. A message includes the topic to which it belongs and the payload.

The broker distributes the message to the two clients that are subscribed to the **sensor1/altitude** topic: the iOS smartphone and the Android tablet.

Publishers and subscribers are decoupled in space because they don't know each other. Publishers and subscribers don't have to run at the same time. The publisher can publish a message and the subscriber can receive it later. In addition, the publish operation isn't synchronized with the receive operation. A publisher requests the broker to publish a message and the different clients that have subscribed to the appropriate topic can receive the message at different times. The publisher can send messages as an asynchronous operation to avoid being blocked until the clients receive the messages. However, it is also possible to send a message to the broker as a synchronous operation with the broker and to continue the execution only after the operation was successful. In most cases, we will want to take advantage of asynchronous operations.

A publisher that requires sending a message to hundreds of clients can do it with a single publish operation to a broker. The broker is responsible for sending the published message to all the clients that have subscribed to the appropriate topic. Because publishers and subscribers are decoupled, the publisher doesn't know whether there is any subscriber that is going to listen to the messages it is going to send. Hence, sometimes it is necessary to make the subscriber become a publisher too and to publish a message indicating that it has received and processed a message. The specific requirements depend on the kind of solution we are building. MQTT offers many features that make our lives easier in many of the scenarios we have been analyzing.

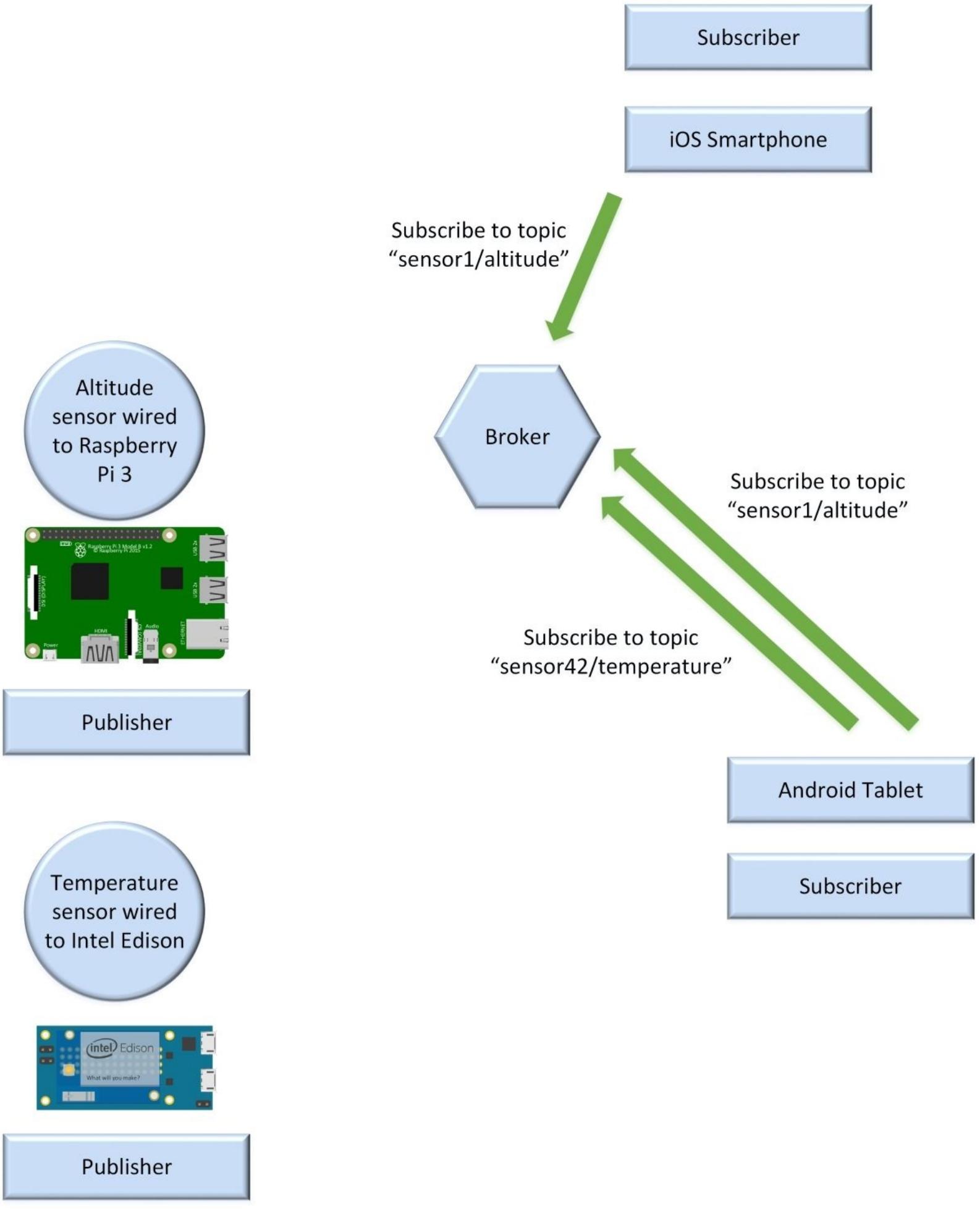
Working with message filtering

The broker has to make sure that subscribers only receive the messages they are interested in. It is possible to filter messages based on different criteria in a publish-subscribe pattern. We will focus on analyzing **topic-based filtering**, also known as subject-based filtering.

Consider that each message belongs to a topic. When a publisher requests the broker to publish a message, it must specify both the topic and the message. The broker receives the message and delivers it to all the subscribers that have subscribed to the topic to which the message belongs.

The broker doesn't need to check the payload for the message to deliver it to the corresponding subscribers; it just needs to check the topic for each message that has arrived and needs to be filtered before publishing it to the corresponding subscribers.

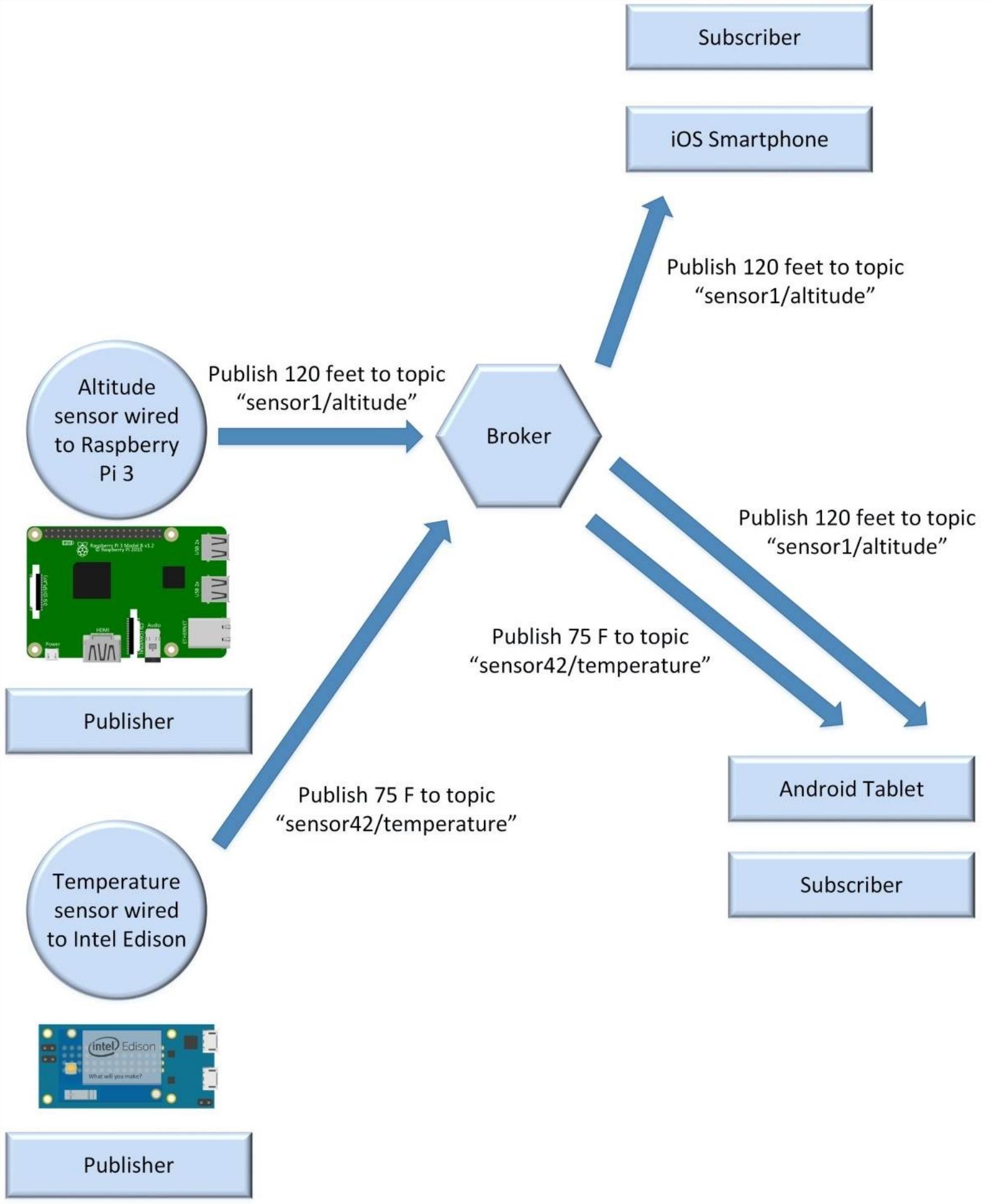
A subscriber can subscribe to more than one topic. In this case, the broker has to make sure that the subscriber receives the messages that belong to all the topics to which it has subscribed. It is easy to understand how things work with another simple diagram. The following diagram shows two future publishers that haven't published any messages yet, a broker and two subscribers connected to the broker:



A Raspberry Pi 3 board with an altitude sensor wired to it and an Intel Edison board with a temperature sensor wired to it will be two publishers. An iOS smartphone and an Android tablet are two subscribers that establish a connection to the broker.

The iOS smartphone indicates to the broker that it wants to subscribe to all the messages that belong to the **sensor1/altitude** topic. The Android tablet indicates to the broker that it wants to subscribe to all the messages that belong to any of the following two topics: **sensor1/altitude** and **sensor42/temperature**. Hence, the Android tablet is subscribed to two topics while the iOS smartphone is subscribed to just one topic.

The following diagram shows what happens after the two publishers connect and publish messages to different topics through the broker:



The Raspberry Pi 3 board publishes a message with **120 feet** as the payload and **sensor1/altitude** as the topic. The board, that is, the publisher, sends the publish request to the broker. The broker distributes the message to the two clients that are subscribed to the **sensor1/altitude** topic: the iOS smartphone and the Android tablet.

The Intel Edison board publishes a message with **75 F** as the payload and **sensor42/temperature** as the topic. The board, that is, the publisher, sends the publish request to the broker. The broker distributes the message to the only client that is subscribed to the **sensor42/temperature** topic: the Android Tablet. Thus, the Android tablet receives two messages.

Understanding the MQTT puzzle -- clients, servers, and connections

The MQTT broker is known as MQTT server in MQTT 3.1.1, and therefore, we will refer to it as the server. However, we must take into account that the documentation for MQTT servers, tools and client libraries can use the old MQTT broker name to refer to the server.

The MQTT server uses the previously explained topic-based filtering to filter and distribute messages to the appropriate subscribers. There are many MQTT server implementations that provide additional message filtering features by providing custom plug-ins. However, we will focus on the features that are part of the MQTT requirements.

As previously explained, in MQTT the publishers and the subscribers are completely decoupled. Publishers and subscribers are MQTT clients that only establish a connection with the MQTT server. An MQTT client can be both a publisher and a subscriber at the same time, that is, the client can publish messages to specific topics and also receive messages that belong to the topics to which the client has subscribed to.

There are MQTT client libraries available for the most popular programming languages and platforms. One of the most important things that we must consider when we select the MQTT client library is the list of MQTT features that they support and the ones that we need for our solution. Sometimes, we can choose between many libraries for a specific programming language and platform and some of them might not implement all the features. We will use many libraries for diverse programming languages and platforms throughout the book and we will learn how to evaluate whether a library is suitable or not for our needs. Specifically, we will work with Python, Java, JavaScript and Swift.

Any device that has a TCP/IP stack and is capable of using an MQTT library can become an MQTT client, that is, a publisher, a subscriber or both a publisher and a subscriber. The MQTT library makes it possible for the device to talk MQTT on top of TCP/IP and to interact with specific types of MQTT servers. For example, any of the following can become an MQTT client: an Arduino board, a Raspberry Pi 3 board, an Intel Joule board, an iPhone, an iPad, a laptop running Windows, a server running Linux, a Macbook running macOS, and Android tablet, among other devices.

There are many MQTT servers available for the most popular platforms, including Linux, Windows and macOS. Many of them are servers that can work as MQTT servers among other features. An MQTT server might implement only a subset of the MQTT capabilities and might have specific limitations. Hence, it is very important to check all the capabilities we will require in our solution before selecting an MQTT server. As happens with other middleware, we have open source versions, free versions and paid versions. Thus, we also have to make sure we select the appropriate MQTT server based on our budget and our specific needs.

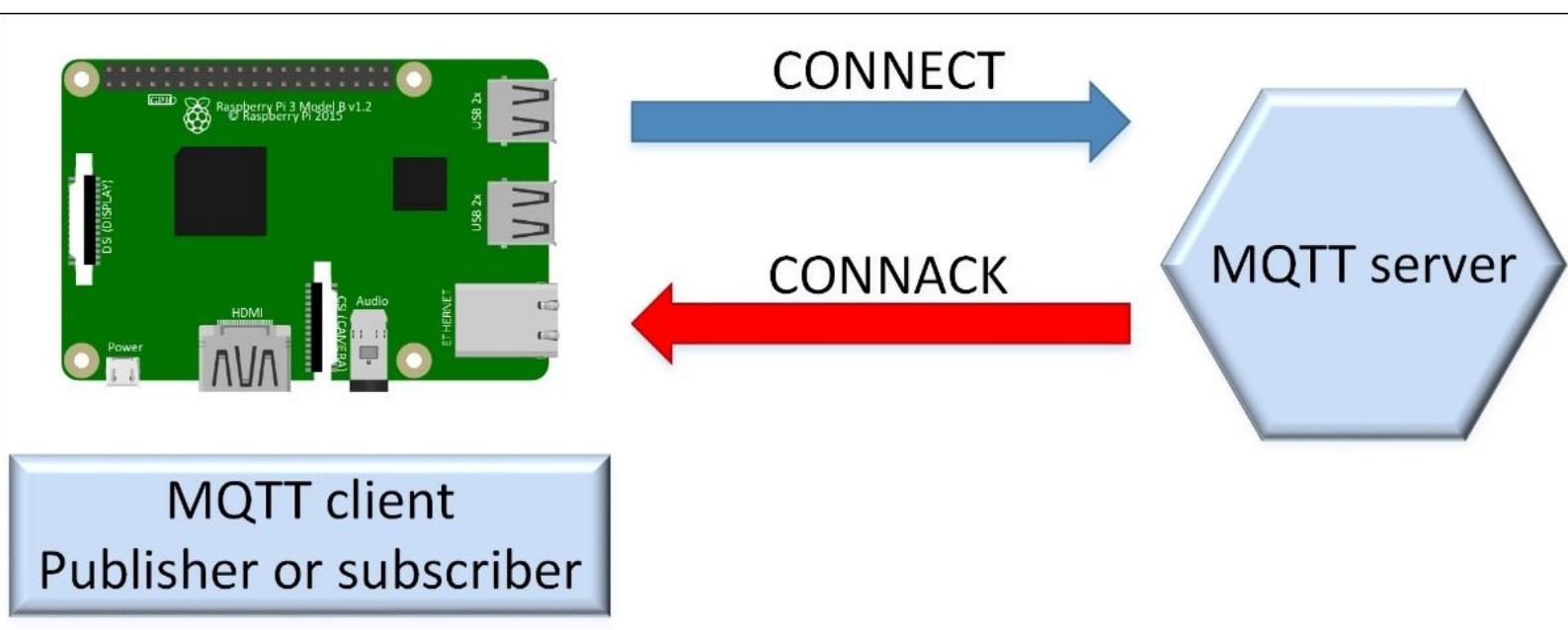
Throughout this book, we will work with the Eclipse Mosquitto MQTT server (<http://www.mosquitto.org>). Mosquitto is an open source MQTT server with an EPL/EDL license that is compatible with MQTT versions 3.1 and 3.1.1. We can take advantage of everything we learn with any

other MQTT server such as **Erlang MQTT Broker (EMQ)**, also known as EMQTT (<http://www.emqtt.io>), and HiveMQ (<http://hivemq.com>), among others. In addition, we might use our knowledge to work with a cloud-based MQTT server such as CloudMQTT (<http://www.cloudmqtt.com>).

The MQTT server is the central hub of the publish/subscribe model we previously analyzed. The MQTT server is responsible of the authentication and authorization of the MQTT clients that will be able to become publishers and/or subscribers after they are authenticated and authorized. So, the first thing that an MQTT client must do is to establish a connection with the MQTT server.

In order to establish a connection, the MQTT client must send a **CONNECT** control packet to the MQTT server with a payload that must include all the necessary information to initiate the connection and proceed with the authentication and authorization. The MQTT server will check the **CONNECT** packet, perform authentication and authorization and send a response to the client with a **CONNACK** control packet that we will analyze in detail after understanding the **CONNECT** control packet. In case the MQTT client sends an invalid **CONNECT** control packet, the server will automatically close the connection.

The following diagram shows the interaction between an MQTT client and an MQTT server to establish a connection.



Tip

After a successful connection has been established between an MQTT client and an MQTT server, the server will keep the connection open until the client loses the connection or sends a **DISCONNECT** control packet to the server to close the connection.

The **CONNECT** control packet must include values for the following fields in the payload and bits for a special flags byte that is included in the control packet. We want to understand the meaning of these fields and flags because we will be able to specify their values when we work with MQTT tools and MQTT client libraries.

- **clientId:** The client identifier, also known as Client ID, is a string that identifies each MQTT client that connects to an MQTT server. Each client that connects to the MQTT server must have a unique `clientId` and the server uses it to identify the state that it holds related to the MQTT session between the client and the server. If a client specifies an empty value as the `ClientId`, the MQTT server must generate a unique `clientId` to identify the client. However, this behavior depends on the value specified for the `CleanSession` field.
- **cleanSession:** The clean session flag is a boolean value that specifies what happens after an MQTT client disconnects from the MQTT server and then reconnects. If `CleanSession` is set to `1` or `true`, the client indicates to the MQTT server that the session will only last as long as the network connection is alive. After the MQTT client disconnects from the MQTT server, any information related to the session is discarded. A new connection from the same MQTT client to the MQTT server will not use the data from the previous session and will be a new clean session. If `CleanSession` is set to `0` or `false`, we will work with a persistent session. In this case, the MQTT server stores all the subscriptions for the MQTT client and when the MQTT client disconnects, the MQTT server stores all the messages that arrive with specific quality of service levels that match the subscriptions that the MQTT client had at the time the disconnection occurred. This way, when the same MQTT client establishes a new connection with the MQTT server, the MQTT client will have the same subscriptions and will receive all the messages that it couldn't receive when it lost the connection. We will dive deep on the quality of service level for the messages and their relationship with the clean session flag or the persistent session option later.

Tip

When the clean session flag is set to `0` or `false`, the client indicates to the server that it wants a persistent session. We just have to remember a clean session is the opposite of a persistent session.

- **UserName:** If the client wants to specify a user name to request authentication and authorization from the MQTT server, it must set the `UserName` flag to `1` or `true` and specify a value for the `UserName` field.
- **password:** If the client wants to specify a password to request authentication and authorization from the MQTT server, it must set the `Password` flag to `1` or `true` and specify a value for the `Password` field.

Tip

We will dedicate the entire next chapter to MQTT security, and therefore, we just mention the fields and the flags that are included in the **CONNECT** control packet.

- **ProtocolLevel:** The protocol level value indicates the MQTT protocol version that the MQTT client requests the MQTT server to use. Remember we will always work with MQTT version 3.1.1.
- **KeepAlive:** `KeepAlive` is a time interval expressed in seconds. If the value for keep alive is not equal to `0`, the MQTT client commits to send control packets to the server within the time specified

for `KeepAlive`. If the MQTT client doesn't have to send any control packet, it must send a `PINGREQ` control packet to the MQTT server, that is, a ping request to tell the MQTT server that the client connection is alive. The MQTT server responds to the `PINGREQ` control packet with a `PINGRESP` response to the MQTT client, that is, a ping response to tell the MQTT client that the connection with the MQTT server is alive. The connection is closed when there is an absence of any of these control packets. If the value for `keep alive` is `0`, the keep alive mechanism is turned off.

- `Will`, `WillQoS`, `WillRetain`, `WillTopic` and `WillMessage`: These flags and fields allow the MQTT client to take advantage of the last will and testament feature of MQTT. If the MQTT client sets the `will` flag to `1` or `true`, it specifies that it wants the MQTT server to store a last will message associated with the session. The `WillQoS` flag specifies the desired quality of service for the last `WillMessage` and the `WillRetain` flag indicates whether this message must be retained when it is published or not. If the MQTT client sets the `will` flag to `1` or `true`, it must specify the topic for the `WillMessage` and the message in the `WillTopic` and `WillMessage` fields. If the MQTT client is disconnected or loses the connection with the MQTT server, the MQTT server will publish the message specified in the `WillMessage` field to the topic indicated in the `WillTopic` field with the chosen quality of service. We will analyze this feature in detail later.

The MQTT server will process a valid `CONNECT` control packet and it will respond with a `CONNACK` control packet. This control packet will include values for the following flags included in the header. We want to understand the meaning of these flags because we will be able to retrieve their values when we work with MQTT tools and MQTT client libraries.

- `SessionPresent`: If the MQTT server received a connection request with the `cleanSession` flag set to `1` or `true`, the value for the `SessionPresent` flag will be `0` or `false` because no stored session will be reused. If the `cleanSession` flag was set to `0` or `false` in the connection request, the MQTT server will work with a persistent session and the value for the `SessionPresent` flag will be `1` or `true` if the server had a persistent session for the client from a previous connection and retrieves it. Otherwise, `SessionPresent` will be `0` or `false`. The MQTT client that wanted to work with a persistent session can use the value of this flag to determine whether it has to request subscription to the desired topics or whether the subscriptions have been restored from the persisted session.
- `ReturnCode`: If the authorization and authentication passed and the connection was successfully established, the value for `ReturnCode` will be `0`. Otherwise, the return code will be different than `0` and the network connection between the client and the server will be closed. The following table shows the possible values for `ReturnCode` with their meaning.

<code>ReturnCode</code> value	Description
<code>0</code>	The connection was accepted
<code>1</code>	The connection was refused because the MQTT server doesn't support the MQTT protocol version requested by the MQTT client in the <code>CONNECT</code> control packet
<code>2</code>	The connection was refused because the <code>clientId</code> (client identifier) specified has been rejected

3 The connection was refused because the network connection was established but the MQTT service isn't available

4 The connection was refused because the user name or password values are malformed

5 The connection was refused because authorization failed

Installing a Mosquitto broker on Ubuntu Linux

Now, we will learn the necessary steps to install a Mosquitto broker, also known as Mosquitto MQTT server on the most popular operating systems: Linux, macOS and Windows. First, we will start with Linux; specifically, we will work with Ubuntu Linux. In case you want to work with a different Linux distribution, you can find details about the installation procedure in the Mosquitto downloads section: <http://mosquitto.org/download>.

Follow these steps to install a Mosquitto broker on Ubuntu Linux. Take into account that you will require root privileges:

1. Open a Terminal window and run the following command to add the Mosquitto repository: **sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa**

After you enter the password, you will see an output similar to the next lines (the temporary file names will be different).

```
gpg: keyring `/tmp/tmpj3a50p1p/secring.gpg' created
gpg: keyring `/tmp/tmpj3a50p1p/pubring.gpg' created
gpg: requesting key 262C4500 from hkp server keyserver.ubuntu.com
gpg: /tmp/tmpj3a50p1p/trustdb.gpg: trustdb created
gpg: key 262C4500: public key "Launchpad mosquitto" imported
gpg: Total number processed: 1
gpg:               imported: 1 (RSA: 1)
OK
```

2. Run the following command to update the packages with the recently added Mosquitto repository.

```
sudo apt-get update
```

You will see an output similar to the next lines.

```
Get:1 http://ppa.launchpad.net/mosquitto-dev/mosquitto-ppa/      ubuntu
xenial InRelease [23,8 kB]
Hit:2 http://security.ubuntu.com/ubuntu xenial-security InRelease
Hit:3 http://archive.ubuntu.com/ubuntu xenial InRelease
Hit:4 http://archive.ubuntu.com/ubuntu xenial-updates InRelease
Get:5 http://ppa.launchpad.net/mosquitto-dev/      mosquitto-ppa/ubuntu
xenial/main amd64 Packages [2.048 B]
Hit:6 http://archive.ubuntu.com/ubuntu xenial-backports InRelease
Get:7 http://ppa.launchpad.net/mosquitto-dev/      mosquitto-ppa/ubuntu
xenial/main i386 Packages [2.048 B]
Get:8 http://ppa.launchpad.net/mosquitto-dev/mosquitto-ppa/
xenial/main Translation-en [1.292 B]
Fetched 29,2 kB in 1s (21,5 kB/s)
Reading package lists... Done
```

3. Now, run the following command to install the package for the Mosquitto server.

```
sudo apt-get install mosquitto
```

You will see an output similar to the next lines. Enter Y and press *Enter* to answer the question and complete the installation process.

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libev4 libuv1 libwebsockets7
The following NEW packages will be installed:
  libev4 libuv1 libwebsockets7 mosquitto
0 upgraded, 4 newly installed, 0 to remove and 299 not upgraded.
8 not fully installed or removed.
Need to get 274 kB of archives.
After this operation, 714 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

The last lines should include a line that says Setting up mosquitto, as shown in the following lines: **Setting up mosquitto (1.4.10-0mosquitto3) ... Processing triggers for libc-bin (2.23-0ubuntu3) ... Processing triggers for systemd (229-4ubuntu7) ... Processing triggers for ureadahead (0.100.0-19) ...**

4. Run the following command to install the Mosquitto client packages that will allow us to run commands to publish messages to topics and subscribe to topic filters.

```
sudo apt-get install mosquitto-clients
```

You will see an output similar to the next lines. Enter Y and press *Enter* to answer the question and complete the installation process.

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libc-ares2 libmosquitto1
The following NEW packages will be installed:
  libc-ares2 libmosquitto1 mosquitto-clients
0 upgraded, 3 newly installed, 0 to remove and 299 not upgraded.
Need to get 138 kB of archives.
After this operation, 332 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

The last lines should include a line that says Setting up mosquitto-clients, as shown in the following lines: **Setting up libmosquitto1:amd64 (1.4.10-0mosquitto3) ... Setting up mosquitto-clients (1.4.10-0mosquitto3) ... Processing triggers for libc-bin (2.23-0ubuntu3) ...**

5. Run the following command to check the status for the mosquitto service that was recently installed.

```
sudo service mosquitto status
```

The first lines of the output should be similar to the following lines with an active (running) status displayed. The details after CGroup indicate the command-line that started the service. The -c option followed by /etc/mosquitto/mosquitto.conf specifies that Mosquitto is using this configuration file.

```
mosquitto.service - LSB: mosquitto MQTT v3.1 message broker
  Loaded: loaded (/etc/init.d/mosquitto; bad; vendor preset:
```

```
enabled)
  Active: active (running)
    Docs: man:systemd-sysv-generator(8)
  CGroup: /system.slice/mosquitto.service
           └─17560 /usr/sbin/mosquitto -c
/etc/mosquitto/mosquitto.conf
```

You can also run the following command to check whether the Mosquitto MQTT server is listening at the default port: 1883.

```
netstat -an | grep 1883
```

The following lines show the results of the previous command that indicate the Mosquitto MQTT server has opened an IPv4 and an IPv6 listen socket on port 1883: **tcp 0 0 0.0.0.0:1883 0.0.0.0:*** **LISTEN** **tcp6 0 0 ::::1883 ::::* LISTEN**

Installing a Mosquitto broker on macOS

Follow these steps to install a Mosquitto broker on macOS (known as OS X before version Sierra):

1. In case you don't have Homebrew installed, open a Terminal window and run the command indicated in the Homebrew homepage, <http://brew.sh>, to install this popular package manager for macOS. The following command will do the job. However, it is convenient to check the Homebrew homepage and check all the detailed instructions that are always updated with the newest versions of macOS that become available. In case you already have Homebrew installed, move to the next step.

```
/usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew  
/install/master/install)"
```

2. Open a Terminal window and run the following command to request Homebrew to install Mosquitto:
brew install mosquitto
3. The following lines show the last messages shown in the Terminal that indicate that Homebrew has installed Mosquitto and the instructions to start the MQTT server.

```
==> Installing mosquitto  
==> Downloading https://homebrew.bintray.com/bottles/mosquitto-  
1.4.10.el_capitan.bottle.tar.gz  
##### 100.0%  
==> Pouring mosquitto-1.4.10.el_capitan.bottle.tar.gz  
==> Caveats  
mosquitto has been installed with a default configuration file.  
You can make changes to the configuration by editing:  
  /usr/local/etc/mosquitto/mosquitto.conf  
To have launchd start mosquitto now and restart at login:  
  brew services start mosquitto  
Or, if you don't want/need a background service you can just run:  
  mosquitto -c /usr/local/etc/mosquitto/mosquitto.conf  
==> Summary  
  /usr/local/Cellar/mosquitto/1.4.10: 32 files, 618.3K
```

4. After the Mosquitto installation has been completed, run the following command in a new Terminal window to launch Mosquitto with the default configuration file. The -c option followed by `/usr/local/etc/mosquitto/mosquitto.conf` specifies that we want to use this configuration file.

```
/usr/local/sbin/mosquitto -c /usr/local/etc/mosquitto/mosquitto.conf
```

The following is the output after you run the previous command:

```
Gastons-MacBook-Pro:~ gaston$ /usr/local/sbin/mosquitto -c  
/usr/local/etc/mosquitto/mosquitto.conf  
1482518967: mosquitto version 1.4.10          (build date 2016-08-31  
20:09:41+0100) starting  
1482518967: Config loaded from  
/usr/local/etc/mosquitto/mosquitto.conf.  
1482518967: Opening ipv4 listen socket on port 1883.  
1482518967: Opening ipv6 listen socket on port 1883.
```

The last lines indicate the Mosquitto MQTT server has opened an IPv4 and an IPv6 listen socket on port 1883. Leave the Terminal window opened because we need Mosquitto running on the local computer to work with the next examples.

Installing a Mosquitto broker on Windows

Follow these steps to install a Mosquitto broker on Windows. Take into account that you will require Windows Vista or higher (Windows 7, 8, 8.1, 10 or greater).

1. Download the executable file listed under **Binary Installation** and **Windows** that provides the native build in the Mosquitto downloads web page: <http://mosquitto.org/download>. For Mosquitto 1.4.10, the file name is `mosquitto-1.4.10-install-win32.exe`. You must click or tap on the file name and you will be redirected to the Eclipse repository with many mirror options, including a default recommendation, from which you can download the executable file.
2. Run the previously downloaded executable file and the Mosquitto Setup Wizard will display its Welcome dialog box. Click **Next >** to continue. The setup wizard will display the dependencies that you must install: OpenSSL and pthreads. The dialog box will display the links that you can use to download and run installers for these two requirements.
3. If you don't have Win32 OpenSSL v1.0.2j Light installed on Windows, go to the Win32 OpenSSL web page: <http://slproweb.com/products/Win32OpenSSL.html> and download the **Win32 OpenSSL v1.0.2k Light** file listed in a table under **Download Win32 OpenSSL**. Do not download the Win64 version because you will need the Win32 version to make it possible for Mosquitto to have its dependencies. If you already have **Win32 OpenSSL v1.0.2k Light** installed, advance to step number 7. For **Win32 OpenSSL v1.0.2k Light**, the file name is `Win32OpenSSL_Light-1_0_2j.exe`. Run the downloaded executable file and OpenSSL Light (32-bit) will display its Welcome dialog box. Click **Next >** to continue.
4. The setup wizard will display the License Agreement. Read it, select **I Accept the agreement** and click **Next >**. Select the folder in which you want to install OpenSSL Light (32-bit) in case you don't want to use the default folder. Remember the folder you specify because you will need to copy a few DLL files from this folder later. The default folder is `C:\OpenSSL-Win32`.
5. Click **Next >** to continue, specify a different Start Menu folder if necessary and click **Next >**. Select **The OpenSSL binaries (/bin) directory** as the desired option to **Copy OpenSSL DLLs to**. This way, the installation will copy the DLLs to a `bin` sub-folder within the previously specified folder, by default, `C:\OpenSSL-Win32\bin`.
6. Click **Next >** to continue. Review the selected installation options and click **Install** to complete the installation for **OpenSSL Light (32-bit)**. Finally, consider a donation to the Win32 OpenSSL project and click **Finish** to exit setup.
7. Go to the following address in your web browser: <ftp://sources.redhat.com/pub/pthreads-win32/dll-latest/dll/x86>. The browser will display many files for this FTP directory. Right click on **pthreadVC2.dll** and save the file in your `Downloads` folder. You will have to copy this DLL to the Mosquitto installation folder later.
8. Now, go back to the Mosquitto Setup window and click **Next >** to continue. By default, Mosquitto will install the files and the Mosquitto Service. Use the default components installation options and click **Next >** to continue.
9. Select the folder in which you want to install Mosquitto in case you don't want to use the default folder. Remember the folder you specify because you will need to copy a few DLL files to this folder later. The default folder is `C:\Program Files (x86)\mosquitto`. Click **Install** to complete the installation. Once the installation has completed click **Finish** to close the Mosquitto Setup

Wizard.

10. Open a File Explorer window and go to the `bin` sub-folder within the folder in which you installed OpenSSL Light (32-bit), by default, `C:\OpenSSL-Win32\bin`.
11. Copy the following two DLLs: `libeay32.dll` and `ssleay32.dll`. Now, go to the folder in which you installed Mosquitto and paste these two DLLs. By default, the Mosquitto installation folder is `C:\Program Files (x86)\mosquitto`. You will need to provide administrator permission to paste the DLLs in the default folder.
12. Open a File Explorer window and go to your `Downloads` folder. Copy the `pthreads` DLL you downloaded in one of the previous steps: `pthreadVC2.dll`. Now, go to the folder in which you installed Mosquitto and paste this DLL. You will need to provide administrator permission to paste the DLL in the default Mosquitto installation folder.
13. Now that all the dependencies are included in the Mosquitto installation folder, it is necessary to run the installation again to make the Mosquitto Setup configure the Windows service. Run the previously downloaded Mosquitto installation executable again. For Mosquitto 1.4.10, the file name is `mosquitto-1.4.10-install-win32.exe`. Make sure you specify the same installation folder as the folder in which you copied the DLLs and that the `Service` component is activated. Click **Next >** many times and click **Install** to complete the configuration for the Windows service. Once the installation has completed click **Finish** to close the Mosquitto Setup Wizard.
14. Open the Services application in Windows and search for the service whose name is **Mosquitto Broker**. Right click on the service name and select **Start**. The Status will change to **Running**. By default, the service is configured to have its Startup Type set to Automatic. If you don't want to automatically start the Mosquitto Broker service, change the Startup Type to Manual. You will have to repeat the step you just did to manually start the service before working with Mosquitto on the Windows computer.

Open a Command Prompt window and run the following command to check whether the Mosquitto MQTT server is listening at the default port: 1883.

```
netstat -an | findstr 1883
```

The following lines show the results of the previous command that indicate the Mosquitto MQTT server has opened an IPv4 and an IPv6 listen socket on port 1883: **TCP 0.0.0.0:1883 0.0.0.0:0 LISTENING** **TCP [::]:1883 [::]:0 LISTENING**

Subscribing to topics

A drone is an IoT device that interacts with many sensors and actuators, including digital electronic speed controllers linked to engines, propellers and servomotors. A drone is also known as an **Unnamed Aerial Vehicle (UAV)**. Let's imagine that we have to monitor many drones. Specifically, we have to display their altitude and the speed for each of their servomotors. Not all of the drones have the same number of engines, propellers and servomotors. We have to monitor quadcopters (4 propellers), hexacopters (6 propellers) and octocopters (8 propellers).

Each drone will publish its altitude every two seconds to the following topic:
`sensors/dronename/altitude`, where `dronename` must be replaced by the name assigned to each drone. For example, the drone named `drone01` will publish its altitude values to the `sensors/drone01/altitude` topic and the drone named `drone20` will use the `sensors/drone20/altitude` topic.

In addition, each drone will publish the speed for each of its rotors every two seconds to the following topic: `sensors/dronename/speed/rotor/rotornumber`, where `dronename` must be replaced by the name assigned to each drone and `rotornumber` must be replaced by the rotor number for which the speed is going to be published. For example, the drone named `drone01` will publish its speed values for its rotor number 1 to the `sensors/drone01/speed/rotor/1` topic.

We will use the `mosquitto_sub` command-line utility included in Mosquitto to generate a simple MQTT client that subscribes to a topic and print all the messages it receives. Open a Terminal in macOS or Linux, or a Command Prompt in Windows, go to the directory in which Mosquitto is installed and run the following command:

```
mosquitto_sub -V mqttv311 -t sensors/drone01/altitude -d
```

The previous command will create an MQTT client that will establish a connection with the local MQTT server and then will make the client subscribe to the topic specified after the `-t` option: `sensors/drone01/altitude`. We specify the version of the MQTT protocol that we want to use when the client establishes the connection with `-V mqttv311`. This way, we indicate the MQTT server that we want to use MQTT version 3.1.1. We specify the `-d` option to enable debug messages that will allow us to understand what happens under the hoods. We will analyze additional options for the connection and the subscription later.

The Terminal or Command Prompt window will display debug messages similar to the following lines. Take into account that the generated `ClientId` will be different from the one shown after `Client`:
`mosqpub/5449-Gastons-Ma`.

```
Client mosqsub/5449-Gastons-Ma sending CONNECT
Client mosqsub/5449-Gastons-Ma received CONNACK
Client mosqsub/5449-Gastons-Ma sending SUBSCRIBE (Mid: 1, Topic:
sensors/drone01/altitude, QoS: 0)
Client mosqsub/5449-Gastons-Ma received SUBACK
Subscribed (mid: 1): 0
```

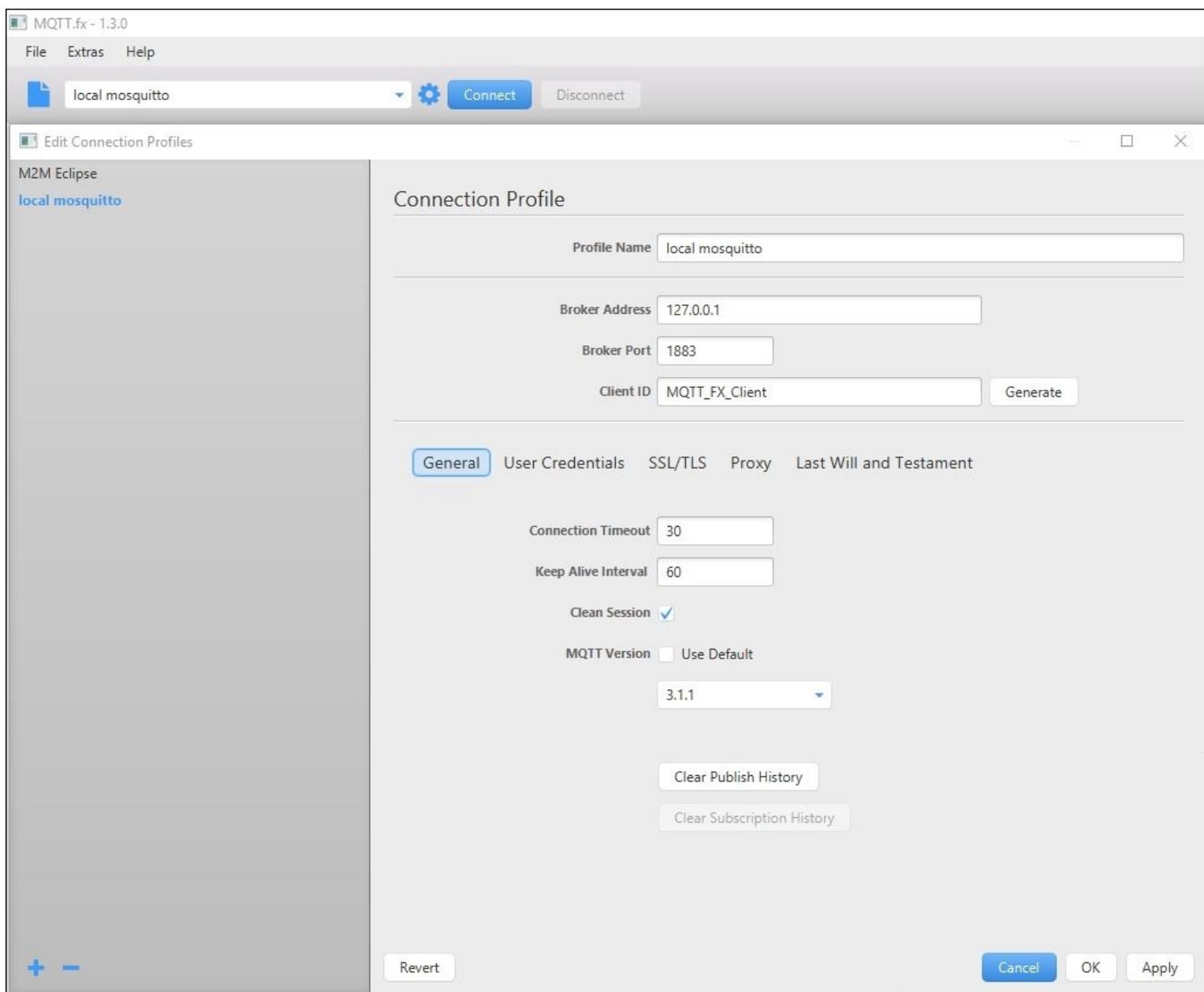
The Terminal or Command Prompt window will display the messages published to the topic to which we subscribed as they arrive from the MQTT server to the MQTT client. Keep the window opened. You will see that the client sends PINGREQ packets to the MQTT server and receives PINQRESP packets from the MQTT server. The following lines show examples of the messages displayed for these packets:

```
Client mosqsub/5449-Gastons-Ma sending PINGREQ
Client mosqsub/5449-Gastons-Ma received PINGRESP
```

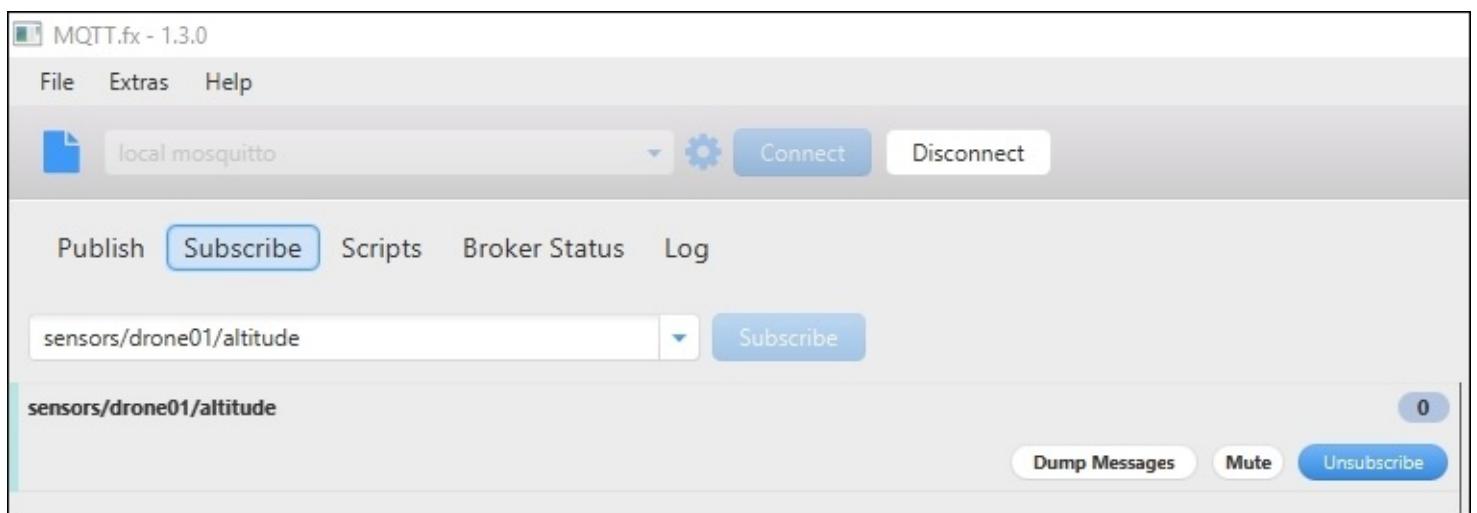
MQTT.fx is a GUI utility implemented with JavaFX that is available for Windows, Linux and macOS. This tool allows us to connect with an MQTT server, subscribe to topic filters, see the received messages and publish messages to topics. You can download the appropriate version for your operating system from the downloads section of the main web page for this utility: <http://www.mqttx.org>.

Now, we will use the MQTT.fx GUI utility to generate another MQTT client that subscribes to the same topic, sensors/drone01/altitude, and displays all the messages it receives. Follow the next steps:

1. Launch MQTT.fx, select **local mosquitto** in the dropdown located at the upper-left corner and click on the configuration icon at the right-hand side of this dropdown and at the left-hand side of the **Connect** button. MQTT.fx will display the **Edit Connection Profiles** dialog box with different options for the connection profile named **local mosquitto**. We analyzed many of these options when we learned about the data that the MQTT client sends to the MQTT server to establish a connection.
2. Make sure the **General** button is pressed and deactivate the **MQTT Version Use Default** checkbox. Make sure **3.1.1** is selected in the dropdown below **MQTT Version**. This way, we indicate to the MQTT server that we want to use MQTT version 3.1.1. Notice that the **Client ID** textbox specifies **MQTT_FX_Client**. This is the `ClientId` value that MQTT.fx will send to the MQTT server (Mosquitto) in the CONNECT control packet. The following screenshot shows the dialog box with the selected options:



3. Click **OK** and then click on the **Connect** button. MQTT.fx will establish a connection with the local Mosquitto server. Notice that the **Connect** button is disabled and the **Disconnect** button is enabled because the client is connected to the MQTT server.
4. Click **Subscribe** and enter `sensors/drone01/altitude` in the dropdown at the left-hand side of the **Subscribe** button. Then, click the **Subscribe** button. MQTT.fx will display a new panel at the left-hand side with the topic to which we have subscribed, as shown in the next screenshot.



In case you don't want to work with the MQTT.fx utility, you can run another `mosquitto_sub` command to generate another MQTT client that subscribes to the topic and print all the messages it receives. You just need to open another Terminal in macOS or Linux, or another Command Prompt in Windows, go to the directory in which Mosquitto is installed and run the following command again. In this case, it isn't necessary to specify the `-d` option.

```
mosquitto_sub -v mqttv311 -t sensors/drone01/altitude
```

Now, we have two MQTT clients subscribed to the same topic: `sensors/drone01/altitude`. Now, we will understand what happens under the hoods when a client subscribes to a topic.

The MQTT client sends a `SUBSCRIBE` packet to the MQTT server with a packet identifier (`PacketId`) in the header and one or more pairs of a topic filter and the desired quality of service level in the payload.

Note

The quality of service is known as QoS.

Hence, a single `SUBSCRIBE` packet can request the MQTT server to subscribe a client to many topics. The `SUBSCRIBE` packet must include at least one topic filter and QoS pair to comply with the protocol.

In the two cases in which we requested a subscription, we used a specific topic name as the value for the topic filter, and therefore, we request the MQTT server to subscribe to a single topic. We will learn about the usage of wildcards in topic filters later.

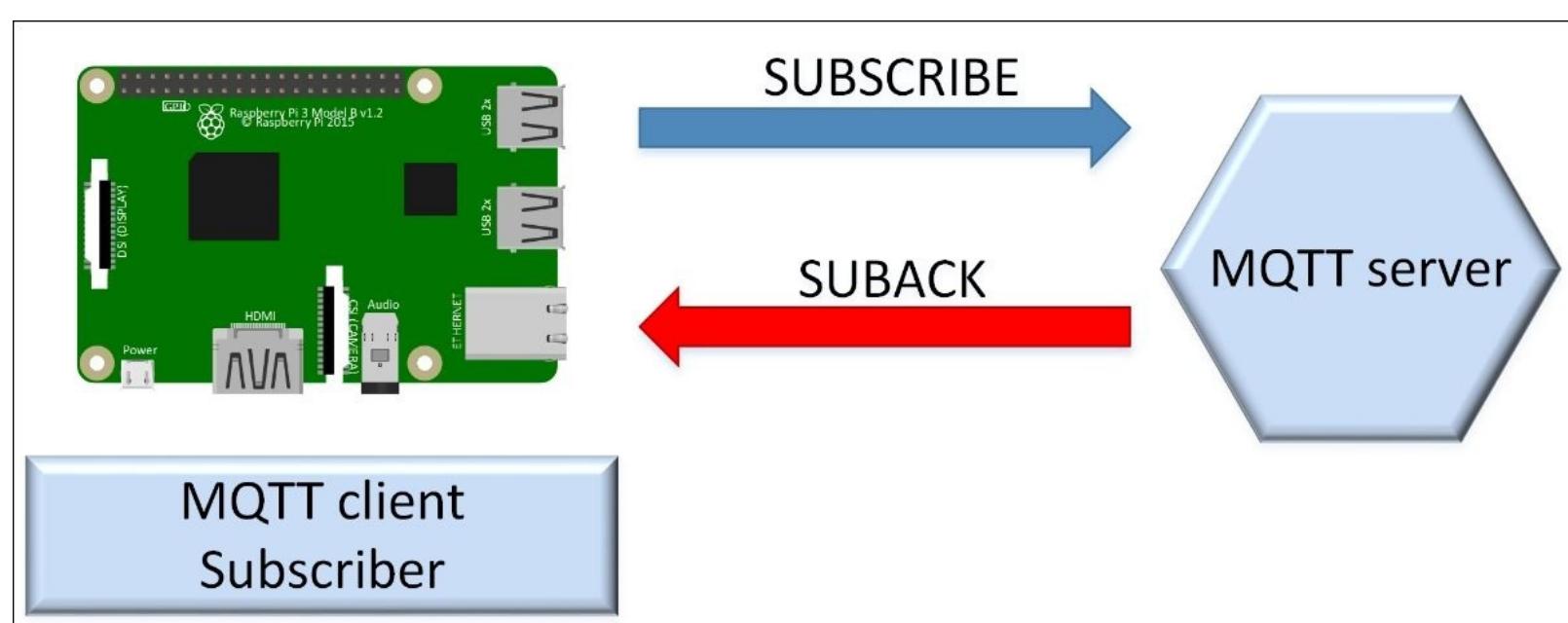
We used the default options, and therefore, the quality of service requested is the default level of 0. We will dive deep on the QoS levels later. Now, we will stay focused on the simplest subscription cases. If the QoS level is equal to 0, the value for the `PacketId` field will be 0. In case the QoS level is equal to 1 or 2, the packet identifier will have a number value to identify the packet and make it possible to identify the responses related to this packet.

The MQTT server will process a valid **SUBSCRIBE** packet and it will respond with a **SUBACK** packet that indicates the subscribe acknowledgement and confirms the receipt and processing of the **SUBSCRIBE** packet. The **SUBACK** packet will include the same packet identifier (Packet ID) in the header that was received in the **SUBSCRIBE** packet. The **SUBACK** packet will include one return code for each pair of a topic filter and the desired QoS level received in the **SUBSCRIBE** packet. The number of return codes will match the number of topic filters included in the **SUBSCRIBE** packet. The following table shows the possible values for these return codes. The first three return codes indicate a successful subscription and each value specifies the maximum QoS that can be delivered based on the requested QoS and the MQTT server possibilities to grant it.

ReturnCode value	Description
0	Successfully subscribed with a maximum QoS of 0
1	Successfully subscribed with a maximum QoS of 1
2	Successfully subscribed with a maximum QoS of 2
128	Failed to subscribe

If the subscription was successful, the MQTT server will start sending every published message that matches the topic filters specified in the subscription to the MQTT client with the QoS specified in the return code.

The following diagram shows the interaction between an MQTT client and an MQTT server to subscribe to one or many topic filters.



Publishing messages

We will use the `mosquitto_pub` command-line utility included in Mosquitto to generate a simple MQTT client that publishes a message to a topic. Open a Terminal in macOS or Linux, or a Command Prompt in Windows, go to the directory in which Mosquitto is installed and run the following command:

```
mosquitto_pub -V mqttv311 -t sensors/drone01/altitude -m "10 f" -d
```

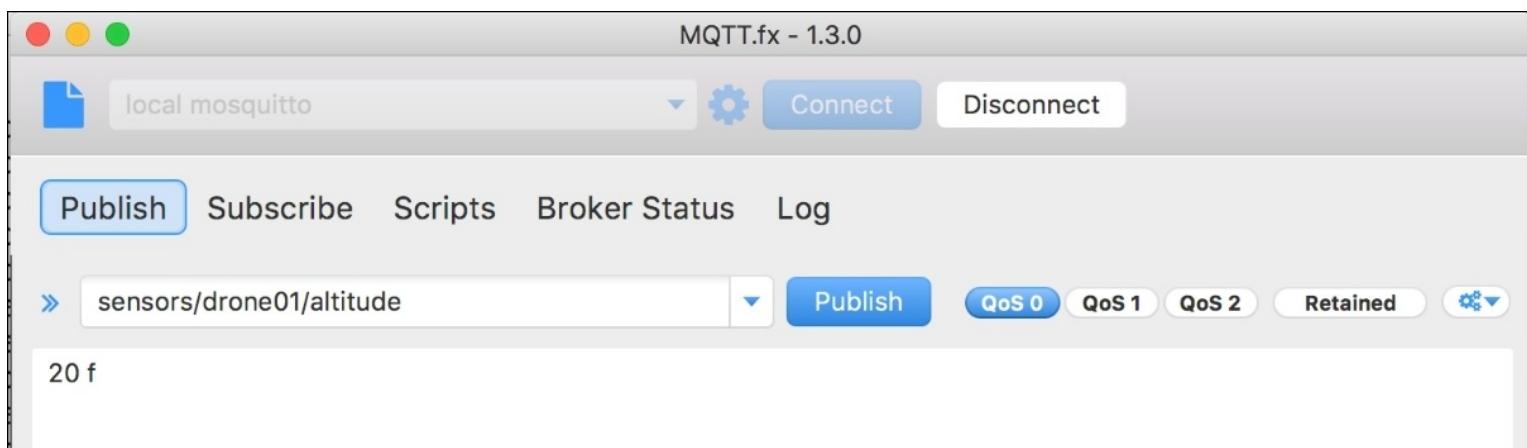
The previous command will create an MQTT client that will establish a connection with the local MQTT server and then will make the client publish a message to the topic specified after the `-t` option: `sensors/drone01/altitude`. We specify the payload for the message after the `-m` option: `"10 f"`. We specify the version of the MQTT protocol that we want to use when the client establishes the connection with `-V mqttv311`. This way, we indicate the MQTT server that we want to use MQTT version 3.1.1. We specify the `-d` option to enable debug messages that will allow us to understand what happens under the hoods. We will analyze additional options for the connection and the publication later.

The Terminal or Command Prompt window will display debug messages similar to the following lines. Take into account that the generated `ClientId` will be different from the one shown after Client:`mosqpub/5303-Gastons-Ma`. After publishing the message, the client disconnects.

```
Client mosqpub/5308-Gastons-Ma sending CONNECT
Client mosqpub/5308-Gastons-Ma received CONNACK
Client mosqpub/5308-Gastons-Ma sending PUBLISH (d0, q0, r0, m1, 'drones/altitude',
... (5 bytes))
Client mosqpub/5308-Gastons-Ma sending DISCONNECT
```

Now, we will use the MQTT.fx GUI utility to generate another MQTT client that publishes another message to the same topic, `sensors/drone01/altitude`. Follow the next steps:

1. Go to the MQTT.fx window in which you established a connection and subscribed to a topic.
2. Click **Publish** and enter `sensors/drone01/altitude` in the dropdown at the left-hand side of the **Publish** button.
3. Enter the following text in the textbox below the **Publish** button: `20 f`, as shown in the following screenshot:



4. Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic.

In case you don't want to work with the MQTT.fx utility, you can run another `mosquitto_pub` command to generate another MQTT client that publishes a message to the topic. You just need to open another Terminal in macOS or Linux, or another Command Prompt in Windows, go to the directory in which Mosquitto is installed and run the following command.

```
mosquitto_pub -V mqttv311 -t sensors/drone01/altitude -m "20 f"
```

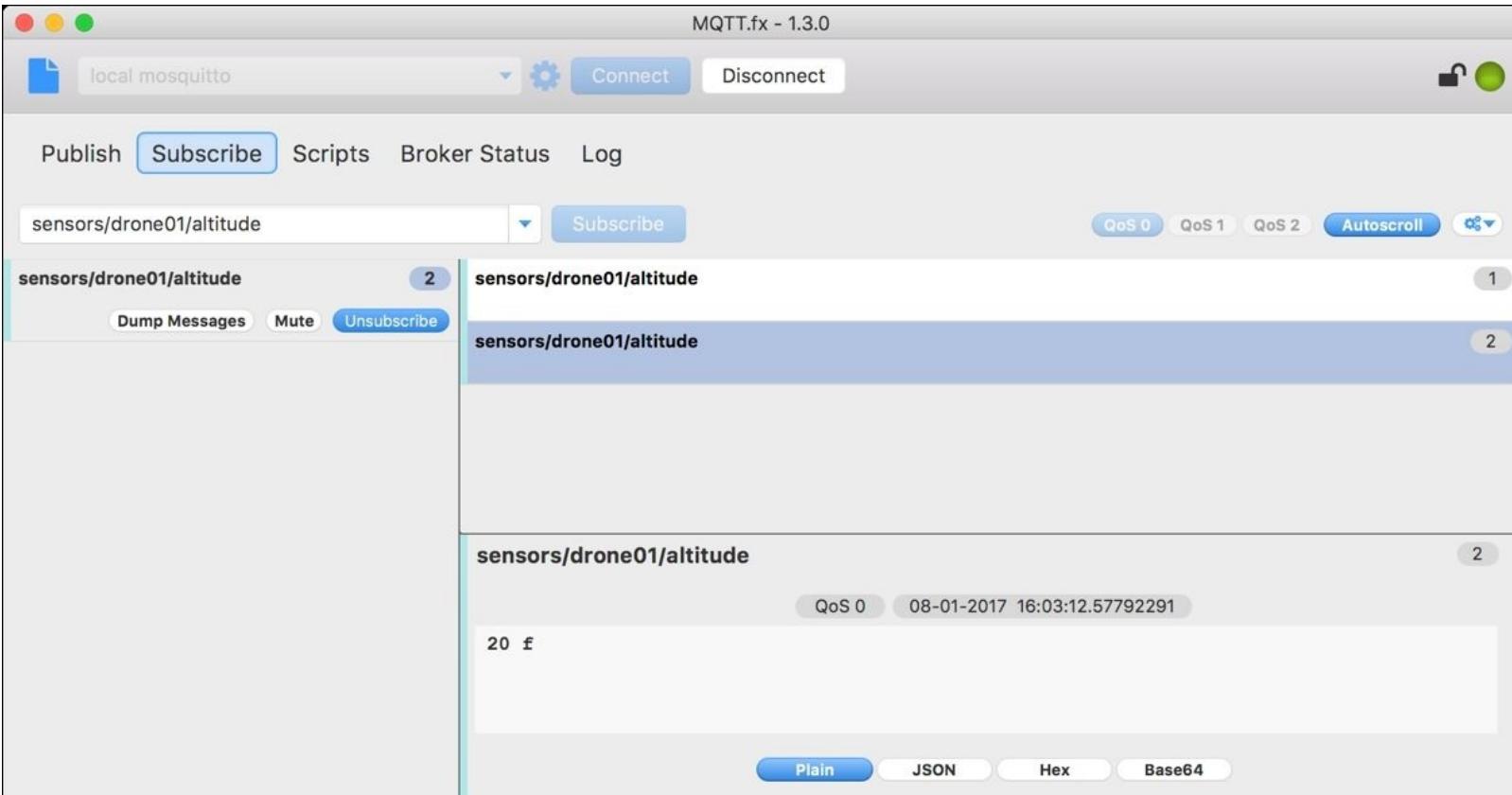
Now, go back to the Terminal or Command Prompt window in which you executed the `mosquitto_sub` command and subscribed to the `sensors/drone01/altitude` topic. You will see lines similar to the following ones:

```
Client mosqsub/5532-Gastons-Ma received PUBLISH (d0, q0, r0, m0,
'sensors/drone01/altitude', ... (4 bytes))
10 f
Client mosqsub/5532-Gastons-Ma received PUBLISH (d0, q0, r0, m0,
'sensors/drone01/altitude', ... (4 bytes))
20 f
```

If we clean up the debug messages that start with the `Client` prefix, we will see just the next two lines. These lines show the payloads for the two messages that we received as a result of our subscription to the `sensors/drone01/altitude` topic.

```
10 f
20 f
```

Go to the MQTT.fx window and click **Subscribe**. You will see **2** at the right-hand side of the title for the topic filter used to subscribe in the panel located at the left-hand side of the window. MQTT.fx is telling you that you have received two messages in the `sensors/drone01/altitude` topic. Click on this panel and MQTT.fx will display all the received messages at the right-hand side of the panel. MQTT.fx will display a number at the right-hand side of each message to specify the message number since we started the subscription to the topic filter. Click on each message and MQTT.fx will display the QoS level for the message (0), the date and time that it was received and the payload for the message in the default plain string format. The following picture shows the payload for the second message that has been received by the subscriber generated by MQTT.fx.



We created two publishers and each of them published a message to the same topic: `sensors/drone01/altitude`. The two subscribers for this topic received both messages. Now, we will understand what happens under the hoods when a client publishes a message to a topic.

The MQTT client that has already established a connection sends a `PUBLISH` packet to the MQTT server with a header that includes the following fields and flags. We want to understand the meaning of these fields and flags because we will be able to specify some of their values when we work with MQTT tools and MQTT client libraries.

- **PacketId:** If the QoS level is equal to 0, the value for this field will be 0 or it won't be present. In case the QoS level is equal to 1 or 2, the packet identifier will have a number value to identify the packet and make it possible to identify the responses related to this packet.
- **Dup:** If the QoS level is equal to 0, the value for this field will be 0. In case the QoS level is equal to 1 or 2, the MQTT client library or the MQTT server can resend a message that was previously published by the client when the subscribers haven't acknowledged the first message. Whenever there is an attempt to resend a message that has already been published, the value for the `Dup` flag must be `1` or `true`.
- **QoS:** Specifies the QoS level for the message. We will dive deep on the quality of service level for the messages and their relationship with many other flags later. So far, we have been working with QoS level 0.
- **Retain:** If the value for this flag is set to 1 or true, the MQTT server will store the message with its specified QoS level. Whenever new MQTT clients subscribe to a topic filter that matches the topic

for the stored or retained message, the last stored message for this topic will be sent to the new subscriber. If the value for this flag is set to 0 or false, the MQTT server won't store the message and won't replace a retained message with the same topic in case there is one message retained for this topic.

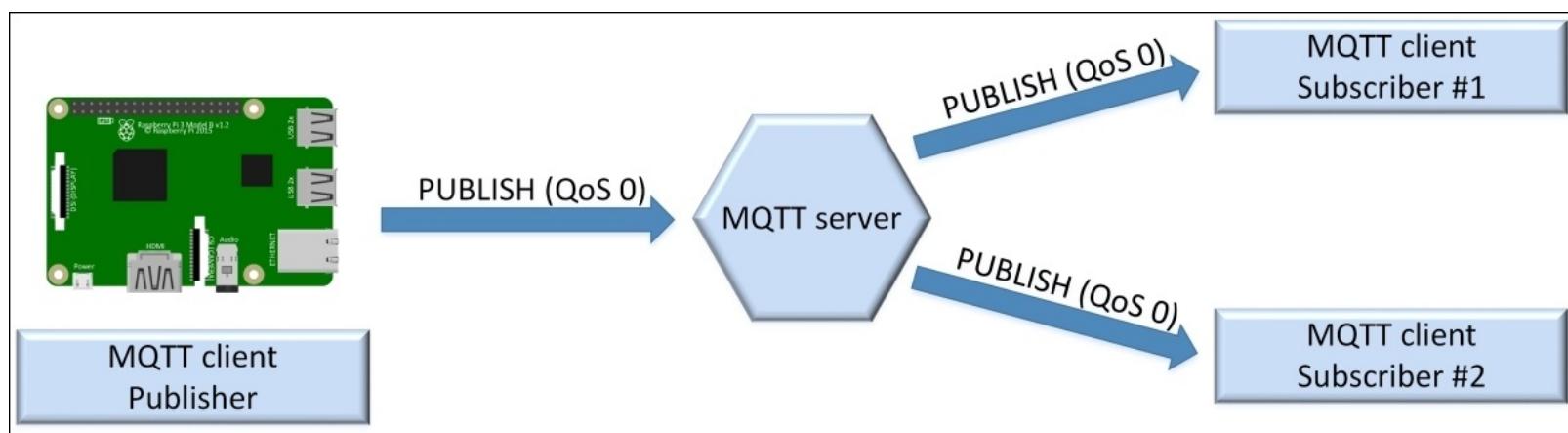
- **TopicName:** A string with the topic name to which the message must be published. Topic names have a hierarchy structure where slashes (/) are used as delimiters. In our examples, the value for TopicName was "sensors/drone01/altitude". We will analyze best practices for topic names later.

Tip

The payload contains the actual message that the MQTT client wants the MQTT server to publish. MQTT is data agnostic, and therefore, we can send any binary data and we don't have restrictions such as those imposed by JSON or XML. Of course, we can use these or others to organize the payloads if we wish. In our examples, we sent a string that included a number that represented the altitude followed by a space and an "f" that indicates the unit of measure is feet.

The MQTT server will read a valid PUBLISH packet and it will respond with a packet only for QoS levels greater than 0. In case the QoS level is 0, the MQTT will not respond. The MQTT server will determine all the subscribers whose subscribed topic matches the topic name specified for the message and the server will publish the message to these clients.

The following diagram shows the interaction between an MQTT client and an MQTT server to publish a message with a QoS level of 0.

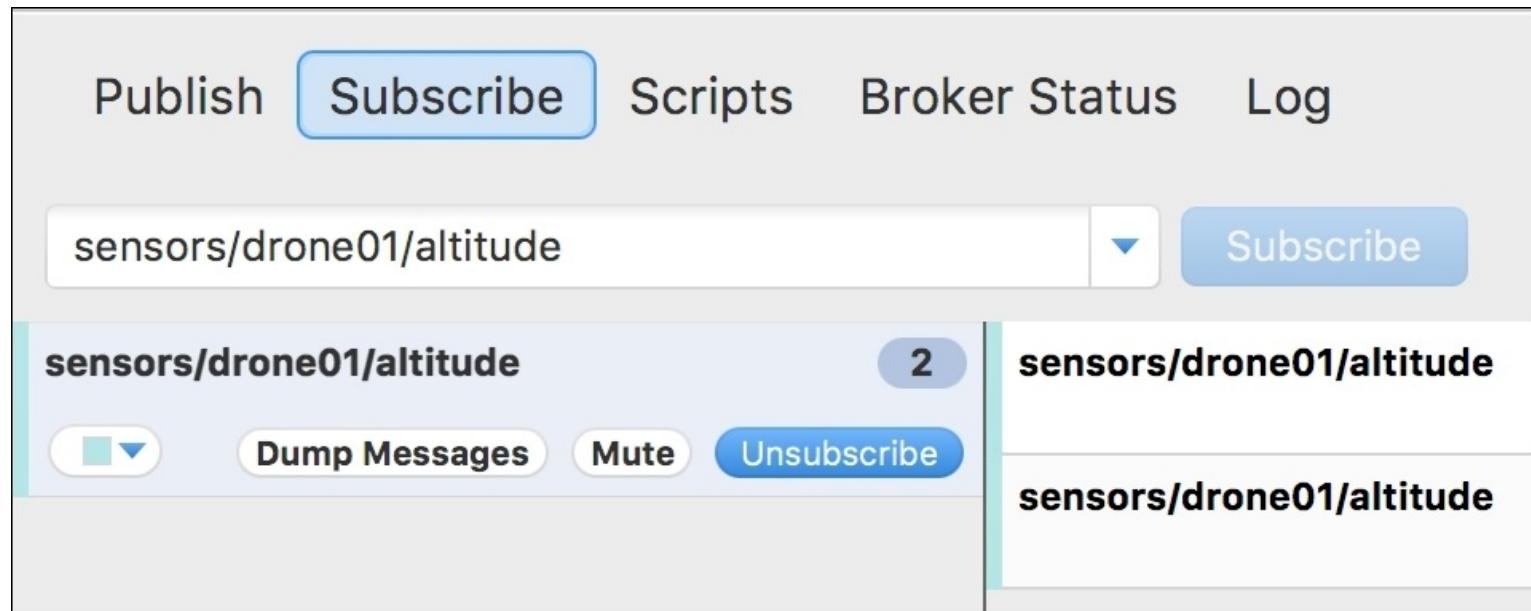


The other QoS levels have a different flow with additional interaction between the publisher and the MQTT server and increased overheads that we will analyze later.

Unsubscribing from topics

Whenever we don't want a subscriber to receive more messages whose destination topic name matches one or more topic filters, the subscriber can send a request to unsubscribe to a list of topic filters to the MQTT server. Obviously, unsubscribing from topic filters is the opposite operation of subscribing to topic filters. We will use the MQTT.fx GUI utility to unsubscribe the MQTT client from the sensors/drone01/altitude topic. Follow the next steps:

1. Go to the MQTT.fx window in which you established a connection and subscribed to a topic.
2. Click **Subscribe**.
3. Click on the panel that displays the sensors/drone01/altitude topic name at the left-hand side of the window. Then, click on the **Unsubscribe** button located in this panel. The following screenshot shows this button.



4. MQTT.fx will unsubscribe the client from the sensors/drone01/altitude topic, and therefore, the client won't receive any new message published to the sensors/drone01/altitude topic.

Now, we will use the MQTT.fx GUI utility to use the MQTT client to publish another message to the sensors/drone01/altitude topic. Follow the next steps:

1. Go to the MQTT.fx window in which you established a connection and subscribed to a topic.
2. Click **Publish** and enter sensors/drone01/altitude in the dropdown at the left-hand side of the **Publish** button.
3. Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic.
4. Enter the following text in the textbox below the **Publish** button: 30 f, as shown in the following screenshot:



The screenshot shows the MQTT.fx application window. At the top, there's a menu bar with File, Extras, and Help. Below it is a toolbar with icons for local mosquitto, Connect, Disconnect, Publish, Subscribe, Scripts, Broker Status, and Log. The main area has tabs for Publish, Subscribe, Scripts, Broker Status, and Log. The Publish tab is active, showing a topic selector dropdown with 'sensors/drone01/altitude', a Publish button, and a message input field containing '30 f'. To the right of the message input are buttons for QoS 0, QoS 1, QoS 2, Retained, and a dropdown menu. On the left, there are two vertical panels labeled 'Switch Four'.

In case you don't want to work with the MQTT.fx utility, you can run a `mosquitto_pub` command to generate another MQTT client that publishes a message to the topic. You just need to open another Terminal in macOS or Linux, or another Command Prompt in Windows, go to the directory in which Mosquitto is installed and run the following command.

```
mosquitto_pub -V mqttv311 -t sensors/drone01/altitude -m "30 f"
```

Now, go back to the MQTT.fx window and click **Subscribe**. The client has unsubscribed from the `sensors/drone01/altitude` topic before we published a new message to this topic, and therefore, this published message isn't displayed.

Go back to the Terminal or Command Prompt window in which you executed the `mosquitto_sub` command and subscribed to the `sensors/drone01/altitude` topic. You will see lines similar to the following ones:

```
Client mosqsub/5532-Gastons-Ma received PUBLISH (d0, q0, r0, m0,
'sensors/drone01/altitude', ... (4 bytes))
30 f
```

This client is still subscribed to the `sensors/drone01/altitude` topic, and therefore, it received the message with the payload of "30 f".

The MQTT client sends an **UNSUBSCRIBE** packet to the MQTT server with a packet identifier (**PacketID**) in the header and one or more topic filters in the payload. The main difference with a **SUBSCRIBE** packet is that it isn't necessary to include the QoS level for each topic filter because the MQTT client just wants to unsubscribe.

Tip

After an MQTT client unsubscribes from one or more topic filters, the MQTT server still keeps the connection opened and the subscriptions to the topic filters that don't match the topic filters specified in the **UNSUBSCRIBE** packet payload will continue working.

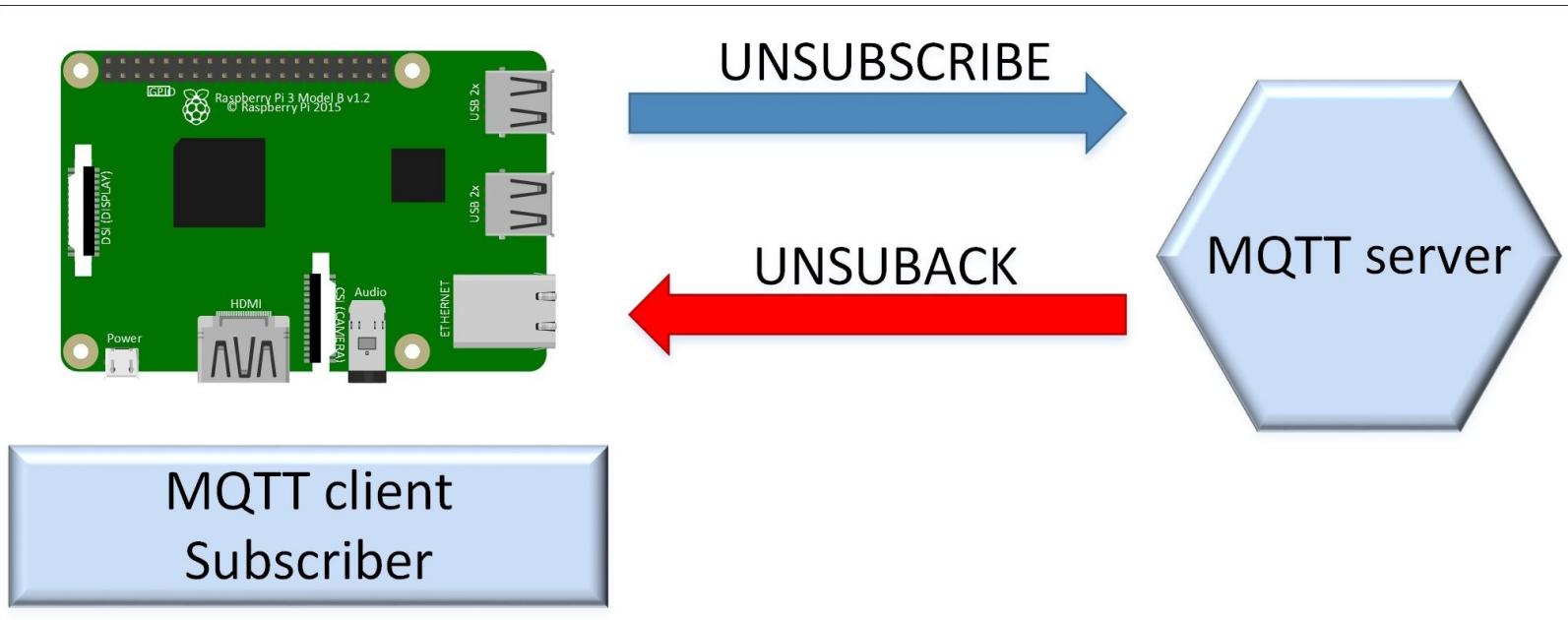
Hence, a single **UNSUBSCRIBE** packet can request the MQTT server to unsubscribe a client from many topics. The **UNSUBSCRIBE** packet must include at least one topic filter in the payload to comply with the protocol.

In the previous example in which we requested the MQTT server to unsubscribe, we used a specific topic name as the value for the topic filter, and therefore, we requested the MQTT server to unsubscribe from a single topic. As previously mentioned, we will learn about the usage of wildcards in topic filters later.

The packet identifier will have a number value to identify the packet and make it possible to identify the response related to this **UNSUBSCRIBE** packet. The MQTT server will process a valid **UNSUBSCRIBE** packet and it will respond with an **UNSUBACK** packet that indicates the unsubscribe acknowledgement and confirms the receipt and processing of the **UNSUBSCRIBE** packet. The **UNSUBACK** packet will include the same packet identifier (**PacketId**) in the header that was received in the **UNSUBSCRIBE** packet.

The MQTT will remove any topic filter that matches exactly any of the specified topic filters in the **UNSUBSCRIBE** packet's payload for the specific client that sent the packet. The topic filter match must be exact to be deleted. After the MQTT server deletes a topic filter from the subscription list for the client, the server stops adding new messages to be published to the client. Only messages that have already started delivery to the client with QoS levels of 1 or 2 will be published to the client. In addition, the server might publish existing messages that have been buffered for their distribution to the subscriber.

The following diagram shows the interaction between an MQTT client and an MQTT server to unsubscribe from one or many topic filters.



Working with best practices when creating topics

We already know that MQTT allows us to publish messages on topics. A publisher always has to specify the topic name to which a message will be published. The easiest way to understand topic names in MQTT is to think about them like paths in a file system.

If we have to save dozens of files that have information about different types of sensors for a diverse number of drones, we can create a hierarchy of directories or folders to organize all the files that we will save. We can create a directory named `sensors`, then one sub-directory for each drone, such as `drone01`, and finally a sub-directory with the sensor name, such as `altitude`. The path in macOS or Linux will be `sensors/drone01/altitude` because these operating systems use a forward slash (/) as a delimiter. In Windows, the path will be `sensors\drone01\altitude` because this operating system uses a backslash (\) as a delimiter.

Then, we will save the files that have information about the altitude sensor for the drone named `drone01` in the created path. Instead of saving files in a path, we can think about publishing a message to a path and use the same mechanism we use to organize files in paths to arrange messages in topics.

Instead of directories or folders, a topic has topic levels, specifically, a hierarchy of topic levels and the slashes (/) are used as delimiters, that is, topic level separators. If we use `sensors/drone01/altitude` as the topic name, `sensors` is the first topic level, `drone01` is the second topic level and `altitude` is the third topic level.

Tip

Topic names are case-sensitive, and therefore, `sensors/drone01/altitude` is different from `sensors/Drone01/altitude`, `Sensors/drone01/altitude` and `Sensors/Drone01/Altitude`. In fact, the four strings will be considered as four individual topic names. We must make sure we select a casing scheme for the topic names and we use it for all the topic names and topic filters.

We can use any UTF-8 character in topic names with the exception of the two wildcard characters that we will analyze later: the plus sign (+) and hash (#). Hence, we must avoid + and # in the topic names. However, it is a good practice to restrict the character set to avoid unexpected problems with client libraries. For example, we can avoid accents and characters that are not common in English as we do whenever we build URLs. It is possible to use these characters but be sure that you can face issues when using them.

We should avoid creating topics starting with the dollar sign (\$) because many MQTT servers publish statistics data, related to the servers, in topics that start with \$. Specifically, the first topic level is `$SYS`.

We must maintain consistency when sending messages to different topic names as we do when we save files in different paths. For example, if we want to publish the altitude for a drone named `drone20`, we will use `sensors/drone20/altitude`. We must use the same topic levels that we used for the same goal for `drone01` and just change the drone name from `drone01` to `drone20`. It would be a really bad practice to use a topic with a different structure or inconsistent casing, such as `altitude/sensors/drone20` or

Sensors/Drone20/Altitude. We have to take into account that we can subscribe to multiple topics by using topic filters, and therefore, it is very important to create topic names accordingly.

Understanding wildcards

When we analyzed the subscription operation, we learn that an MQTT client can subscribe to one or more topic filters. In case we specify a topic name as a topic filter, we will only subscribe to a single topic. We can take advantage of the following two wildcards to create topic filters that subscribe to all the topics that match the filter:

- Plus sign (+): It is a single level wildcard that matches any name for a specific topic level. We can use this wildcard instead of specifying a name for any topic level in the topic filter.
- Hash (#): It is a multi level wildcard that we can use only at the end of the topic filter, as the last level and matches any topic whose first levels are the same as the topic levels specified at the left-hand side of the # symbol.

For example, in case we want to receive all the messages related to the altitudes for all the drones, we can use the + single level wildcard instead of a specific drone name. We can use the following topic filter: `sensors/+/altitude`.

If we publish messages to the following topics, the subscriber that used the `sensors/+/altitude` topic filter will receive all of them:

- `sensors/drone01/altitude`
- `sensors/drone02/altitude`
- `sensors/superdrone01/altitude`
- `sensors/thereatestdrone/altitude`

The subscriber to the `sensors/+/altitude` topic filter won't receive messages sent to any of the following topics, because they won't match the topic filter.

- `sensors/drone01/speed/rotor/1`
- `sensors/superdrone01/speed/rotor/2`
- `sensors/superdrone01/remainingbattery`

In case we want to receive all the messages related to all the sensors for the drone named `drone01`, we can use the # multi level wildcard after the drone name and the slash (/). We can use the following topic filter: `sensors/drone01/#`.

If we publish messages to the following topics, the subscriber that used the `sensors/drone01/#` topic filter will receive all of them:

- `sensors/drone01/altitude`
- `sensors/drone01/speed/rotor/1`
- `sensors/drone01/speed/rotor/2`
- `sensors/drone01/speed/rotor/3`
- `sensors/drone01/speed/rotor/4`
- `sensors/drone01/remainingbattery`

We used a multi level wildcard, and therefore, no matter the additional topic levels after `sensors/drone01/`, we will receive all of them.

The subscriber to the `sensors/drone01/#` topic filter won't receive messages sent to any of the following topics, because they won't match the topic filter. None of the following has `sensors/drone01/` as a prefix, and therefore, they don't match the topic filter.

- `sensors/drone02/altitude`
- `sensors/superdrone01/altitude`
- `sensors/thereatestdrone/altitude`
- `sensors/drone02/speed/rotor/1`
- `sensors/superdrone02/speed/rotor/2`
- `sensors/superdrone02/remainingbattery`

Obviously, we must be careful when we use any wildcard because we might be subscribing to a huge number of topics with a single topic filter. We have to avoid subscribing to topics aren't of interest to the client to avoid wasting unnecessary bandwidth and the broker server resources.

We will use these wildcards in subscriptions later to analyze how different QoS levels work with MQTT.

Learning about the different Quality of Service levels

Now that we understand how the connection, subscription and publication works in combination with topic names and the topic filters with wildcards, we can dive deep on the QoS levels. So far, we have analyzed how both the subscription and publication worked with a QoS level equal to 0. Now, we will understand what this number means and how things work when we use the other available QoS levels for the publication and the subscription.

Note

Remember that publication involves publishing from the MQTT client to the MQTT server and then from the server to the subscribed client. It is very important to understand that we can publish with a QoS level and subscribe with another QoS level. Hence, there is a QoS level for the publish process between the publisher and the MQTT server and another QoS level for the publish process between the MQTT server and the subscriber. We will use sender and receiver to identify the parties involved in a message delivery for the different QoS levels. In the publish process between the publisher and the MQTT server, the publisher will be the sender and the MQTT server the receiver. In the publish process between the MQTT server and the subscriber, the sender will be the MQTT server and the receiver will be the subscriber.

Based on the QoS level, we have differences in the meaning of a successfully delivered message in the MQTT protocol between the involved parties. The QoS level is the agreement between a sender and a receiver of a message about the guarantees of actually delivering the message. These warranties might include the number of times that a message might arrive and the possibility or not of duplicates. MQTT supports the following three possible QoS levels:

- **0, At most once delivery:** This QoS level offers the same warranty as the underlying TCP protocol. The message is not acknowledged by the receiver or destination. The sender just sends the message to the destination and nothing else happens. The sender neither stores nor schedules new deliveries for any messages that might fail to reach the destination. The key advantage of this QoS level is that it has the lowest possible overhead, compared to the other QoS levels.
- **1, At least once delivery:** This QoS level adds a confirmation requirement to the destination that has to receive the message. This way, QoS level 1 provides a warranty that the message will be delivered at least once to the subscriber. One of the key disadvantages of this QoS level is that it might generate duplicates, that is, the same message might be sent more than once to the same destination. The sender stores the message until it receives an acknowledgement from the subscriber. In case the sender doesn't receive the acknowledgement within a specific time, the sender will publish the message again to the receiver. The final receiver must have the necessary logic to detect duplicates in case they shouldn't be processed twice.
- **2, Exactly once delivery:** This QoS level provides a warranty that the message is delivered only once to the destination. QoS level 2 has the highest overhead, compared to the other QoS levels. This QoS level requires two flows between the sender and the receiver. A message published with QoS level 2 is considered successfully delivered after the sender is sure that it has been successfully received once by the destination.

Sometimes, we just want messages to be delivered with the least possible bandwidth usage, we have a very reliable network and it doesn't matter if for some reason a few messages are lost. In these cases, QoS level 0 is the appropriate choice.

In other cases, the messages are extremely important because they represent commands to control an IoT device, the network is unreliable and we must make sure that the message reaches the destination. In addition, a duplicate command might generate a big problem because we don't want the IoT device to process a specific command twice. In these cases, QoS level 2 is going to be the appropriate choice.

Note

If a publisher works with a QoS level higher than the QoS level specified by the subscriber, the MQTT server will have to downgrade the QoS level to the lowest level that the specific subscriber is using when it publishes the message from the MQTT server to this subscriber. For example, if we use QoS level 2 to publish a message from the publisher to the MQTT server but one subscriber has requested QoS level 1 when making the subscription, the publication from the MQTT server to this subscriber will use QoS level 1.

Working with at least once delivery

First, we will use wildcards to subscribe to a topic filter with QoS level 1 and then we will publish one message to a topic name that will match the topic filter with QoS level 1. This way, we will analyze how both publishing and subscription work with QoS level 1.

We will use the `mosquitto_sub` command-line utility included in Mosquitto to generate a simple MQTT client that subscribes to a topic filter with QoS level 1 and prints all the messages it receives. Open a Terminal in macOS or Linux, or a Command Prompt in Windows, go to the directory in which Mosquitto is installed and run the following command:

```
mosquitto_sub -V mqttv311 -t sensors/+/altitude -q1 -d
```

The previous command will create an MQTT client that will establish a connection with the local MQTT server and then will make the client subscribe to the topic filter specified after the `-t` option: `sensors/+/altitude`. We specify that we want to use QoS level 1 to subscribe to the topic filter with the `-q 1` option. We specify the `-d` option to enable debug messages that will allow us to understand what happens under the hoods and the differences with publishing a message with QoS level 0.

The Terminal or Command Prompt window will display debug messages similar to the following lines. Take into account that the generated `ClientId` will be different from the one shown after `Client mosqsub/1795-Gastons-Ma`. Notice that `QoS: 1` indicates that the subscription is done with QoS level 1.

```
Client mosqsub/1795-Gastons-Ma sending CONNECT
Client mosqsub/1795-Gastons-Ma received CONNACK
Client mosqsub/1795-Gastons-Ma sending SUBSCRIBE (Mid: 1, Topic:
sensors/+/altitude, QoS: 1)
Client mosqsub/1795-Gastons-Ma received SUBACK
Subscribed (mid: 1): 1
```

We will use the `mosquitto_pub` command-line utility included in Mosquitto to generate a simple MQTT client that publishes a message to a topic with QoS level 1 instead of the QoS level 0 that we used when we published messages before. Open a Terminal in macOS or Linux, or a Command Prompt in Windows, go to the directory in which Mosquitto is installed and run the following command:

```
mosquitto_pub -V mqttv311 -t sensors/drone02/altitude -m "65 f" -q 1 -d
```

The previous command will create an MQTT client that will establish a connection with the local MQTT server and then will make the client publish a message to the topic specified after the `-t` option: `sensors/drone02/altitude`. We specify the payload for the message after the `-m` option: `"65 f"`. We specify that we want to use QoS level 1 to publish the message with the `-q 1` option. We specify the `-d` option to enable debug messages that will allow us to understand what happens under the hoods and the differences with publishing a message with QoS level 0.

The Terminal or Command Prompt window will display debug messages similar to the following lines. Take into account that the generated `ClientId` will be different from the one shown after `Client`:

mosqpub/1516-Gastons-Ma. After publishing the message, the client disconnects.

```
Client mosqpub/1516-Gastons-Ma sending CONNECT
Client mosqpub/1516-Gastons-Ma received CONNACK
Client mosqpub/1516-Gastons-Ma sending PUBLISH (d0, q1, r0, m1,
'sensors/drone02/altitude', ... (4 bytes))
Client mosqpub/1516-Gastons-Ma received PUBACK (Mid: 1)
Client mosqpub/1516-Gastons-Ma sending DISCONNECT
```

The previous lines show that the generated MQTT client sends a PUBLISH packet to the MQTT server and then receives a PUBACK package from the server.

Now, go back to the Terminal or Command Prompt window in which you executed the mosquitto_sub command and subscribed to the sensors/+/altitude topic filter. You will see lines similar to the following ones:

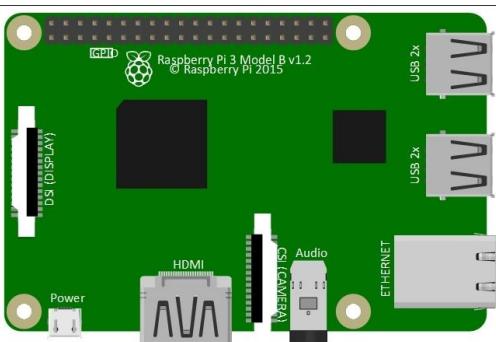
```
Client mosqsub/1795-Gastons-Ma received PUBLISH (d0, q1, r0, m1,
'sensors/drone02/altitude', ... (4 bytes))
Client mosqsub/1795-Gastons-Ma sending PUBACK (Mid: 1)
65 f
```

The previous lines show that the generated MQTT client, that is, the subscriber, received a PUBLISH packet from the MQTT server and then sent a PUBACK package to the server to acknowledge the message. If we clean up the debug messages that start with the Client prefix, we will see just the last line that shows the payloads for the message that we received as a result of our subscription to the sensors/+/altitude topic filter: 65 f.

The MQTT client that has already established a connection, that is, the publisher, sends a PUBLISH packet to the MQTT server with the header we have already described, QoS set to 1, and including a PacketId numeric value that will be unique for this client. At this time, the publisher will consider the PUBLISH packet identified with the PacketId as an unacknowledged PUBLISH packet.

The MQTT server reads a valid PUBLISH packet and it will respond to the publisher with a PUBACK packet with the same PacketId value that has been used for the PUBLISH packet. Once the publisher receives the PUBACK packet, it discards the message and the MQTT server is responsible of publishing it to the appropriate subscribers.

The following diagram shows the interaction between a publisher and an MQTT server to publish a message with a QoS level of 1.



MQTT client
Publisher

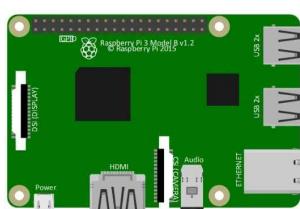
PUBLISH (QoS 1)

PUBACK

MQTT server

The MQTT server can start publishing messages to the appropriate subscribers before it sends the PUBACK packet to the publisher. Hence, when the publisher receives the PUBACK packet from the MQTT server, it doesn't mean that all the subscribers have received the message. It is very important to understand the meaning of this PUBACK packet.

For each subscriber to which the message has to be published, the MQTT server will send a PUBLISH packet and the subscriber must acknowledge the reception of the message by sending a PUBACK packet to the MQTT server. The following diagram shows the interaction between an MQTT server and the subscribers when a message is published with a QoS level of 1.



MQTT client
Publisher

PUBLISH (QoS 1)

MQTT server

PUBLISH (QoS 1)

PUBACK

PUBLISH (QoS 1)

PUBACK

MQTT client
Subscriber #1
QoS 1

MQTT client
Subscriber #2
QoS 1

If the application is able to tolerate duplicates and we have to make sure that the messages arrive at least once to the subscribers, QoS level 1 is an excellent choice. In case there is no way to handle duplicates, we have to use QoS level 2.

Working with exactly once delivery

First, we will use wildcards to subscribe to a topic filter with QoS level 2 and then we will publish one messages to a topic that will match the topic filter with QoS level 2. This way, we will analyze how both publishing and subscription work with QoS level 2.

We will use the `mosquitto_sub` command-line utility included in Mosquitto to generate a simple MQTT client that subscribes to a topic filter with QoS level 1 and prints all the messages it receives. Open a Terminal in macOS or Linux, or a Command Prompt in Windows, go to the directory in which Mosquitto is installed and run the following command:

```
mosquitto_sub -V mqttv311 -t sensors/drone03/# -q 2 -d
```

The previous command will create an MQTT client that will establish a connection with the local MQTT server and then will make the client subscribe to the topic filter specified after the `-t` option: `sensors/drone03/#`. We specify that we want to use QoS level 1 to subscribe to the topic filter with the `-q 2` option. We specify the `-d` option to enable debug messages that will allow us to understand what happens under the hoods and the differences with publishing a message with QoS levels 0 and 1.

The Terminal or Command Prompt window will display debug messages similar to the following lines. Take into account that the generated `ClientId` will be different from the one shown after `Client` `mosqsub/1795-Gastons-Ma`. Notice that `QoS: 2` indicates that the subscription is done with QoS level 2.

```
Client mosqsub/2017-Gastons-Ma sending CONNECT
Client mosqsub/2017-Gastons-Ma received CONNACK
Client mosqsub/2017-Gastons-Ma sending SUBSCRIBE (Mid: 1, Topic: sensors/drone03/#, QoS: 2)
Client mosqsub/2017-Gastons-Ma received SUBACK
Subscribed (mid: 1): 2
```

We will use the `mosquitto_pub` command-line utility included in Mosquitto to generate a simple MQTT client that publishes a message to a topic with QoS level 2 instead of the QoS levels 0 and 1 that we used when we published messages before. Open a Terminal in macOS or Linux, or a Command Prompt in Windows, go to the directory in which Mosquitto is installed and run the following command:

```
mosquitto_pub -V mqttv311 -t sensors/drone03/speed/rotor/1 -m "362 f" -q 2 -d
```

The previous command will create an MQTT client that will establish a connection with the local MQTT server and then will make the client publish a message to the topic specified after the `-t` option: `sensors/drone03/speed/rotor/1`. We specify the payload for the message after the `-m` option: `"362 f"`. We specify that we want to use QoS level 2 to publish the message with the `-q 2` option. We specify the `-d` option to enable debug messages that will allow us to understand what happens under the hoods and the differences with publishing a message with QoS levels 0 and 1.

The Terminal or Command Prompt window will display debug messages similar to the following lines. Take into account that the generated `ClientId` will be different from the one shown after `Client`:

mosqpub/2026-Gastons-Ma. After publishing the message, the client disconnects.

```
Client mosqpub/2026-Gastons-Ma sending CONNECT
Client mosqpub/2026-Gastons-Ma received CONNACK
Client mosqpub/2026-Gastons-Ma sending PUBLISH (d0, q2, r0, m1,
'sensors/drone03/speed/rotor/1', ... (5 bytes))
Client mosqpub/2026-Gastons-Ma received PUBREC (Mid: 1)
Client mosqpub/2026-Gastons-Ma sending PUBREL (Mid: 1)
Client mosqpub/2026-Gastons-Ma received PUBCOMP (Mid: 1)
Client mosqpub/2026-Gastons-Ma sending DISCONNECT
```

The previous lines show that the generated MQTT client, that is, the publisher, had the following packet exchanges with the MQTT server:

1. The publisher sent a PUBLISH packet to the MQTT server.
2. The publisher received a PUBREC packet from the MQTT server.
3. The publisher sent a PUBREL packet to the MQTT server.
4. The publisher received a PUBCOMP packet from the MQTT server.

Now, go back to the Terminal or Command Prompt window in which you executed the `mosquitto_sub` command and subscribed to the `sensors/drone03/#` topic filter. You will see lines similar to the following ones:

```
Client mosqsub/2017-Gastons-Ma received PUBLISH (d0, q2, r0, m1,
'sensors/drone03/speed/rotor/1', ... (5 bytes))
Client mosqsub/2017-Gastons-Ma sending PUBREC (Mid: 1)
Client mosqsub/2017-Gastons-Ma received PUBREL (Mid: 1)
362 f
Client mosqsub/2017-Gastons-Ma sending PUBCOMP (Mid: 1)
```

The previous lines show that the generated MQTT client, that is, the subscriber, had the following packet exchanges with the MQTT server:

1. The subscriber received a PUBLISH packet from the MQTT server.
2. The subscriber sent a PUBREC packet to the MQTT server.
3. The subscriber received a PUBREL packet from the MQTT server.
4. The subscriber sends a PUBCOMP packet to the MQTT server after it successfully received the message with the payload.

If we clean up the debug messages that start with the **Client** prefix, we will see just the last line that shows the payloads for the message that we received as a result of our subscription to the `sensors/drone03/#` topic filter: 362 f.

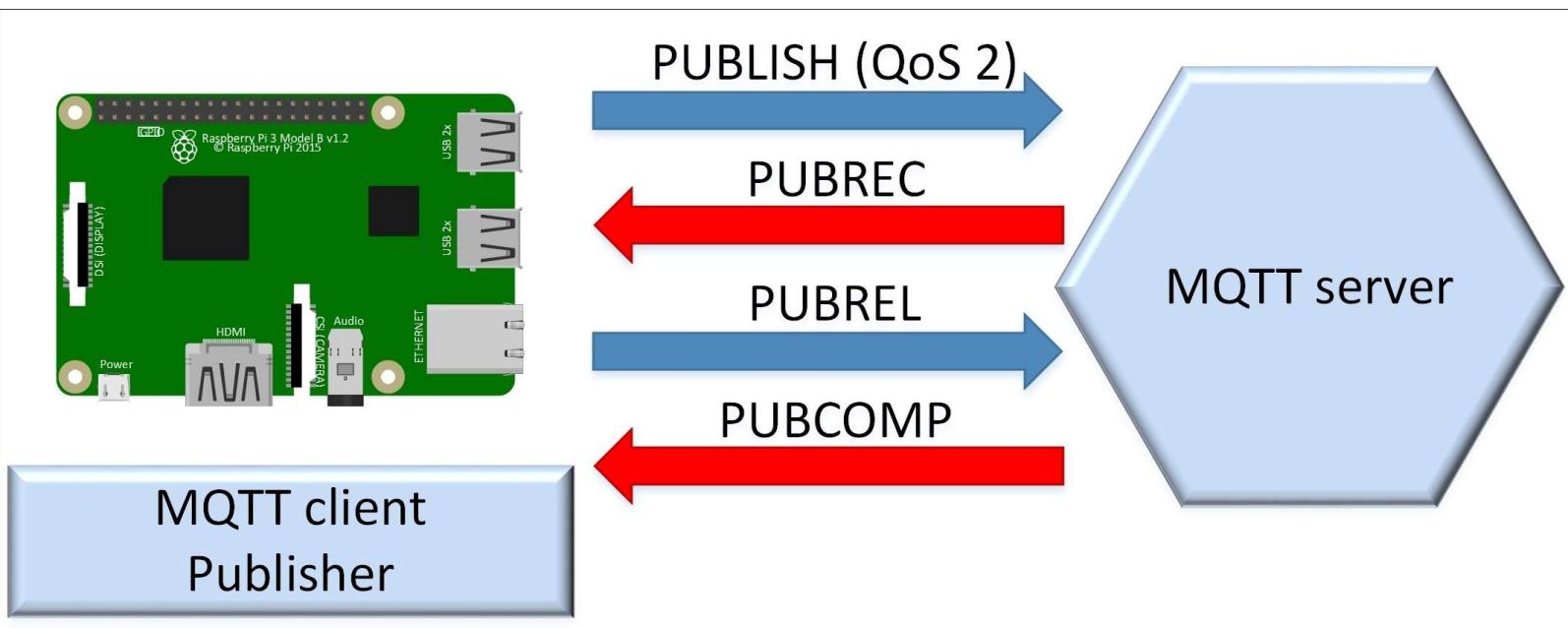
The MQTT client that has already established a connection, that is, the publisher sends a PUBLISH packet to the MQTT server with the header we have already described, QoS set to 2, and including a PacketId numeric value that will be unique for this client. At this time, the publisher will consider the PUBLISH packet identified with the PacketId as an unacknowledged PUBLISH packet.

The MQTT server reads a valid PUBLISH packet and it will respond to the publisher with a PUBREC

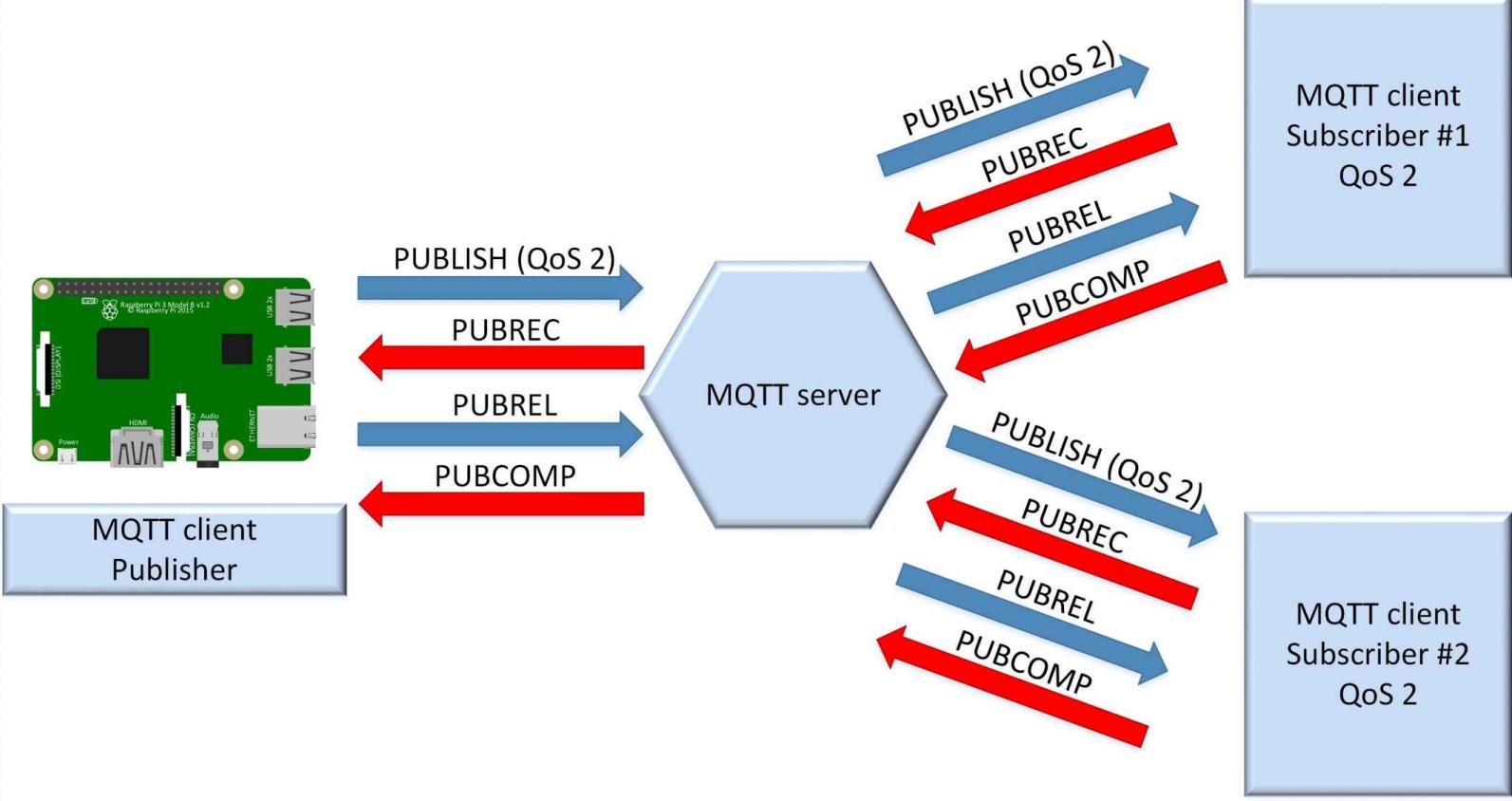
packet with the same `PacketId` value that has been used for the `PUBLISH` packet. The `PUBREC` packet indicates the MQTT server accepted the ownership of the message. Once the publisher receives the `PUBREC` packet, it discards the message and it will store the `PacketId` related to the message and the `PUBREC` packet.

The publisher sends a `PUBREL` packet to the MQTT server as a response to the received `PUBREC` packet. This `PUBREL` packet will be considered unacknowledged until it receives the `PUBCOMP` packet related to the `PacketId` from the MQTT server. Finally, the MQTT server sends a `PUBCOMP` packet with the `PacketId` to the publisher and at this point, both the publisher and the MQTT server are sure that the message has been successfully delivered.

The following diagram shows the interaction between a publisher and an MQTT server to publish a message with a QoS level of 2.



For each subscriber with QoS level 2 to which the message has to be published, the MQTT server will send a `PUBLISH` packet and the same packet exchange that we have analyzed between the publisher and the MQTT server will happen between the MQTT server and the subscriber. However, in this case, the MQTT server is the one that acts as a publisher and starts the flow. The following diagram shows the interaction between an MQTT server and the subscribers when a message is published with a QoS level of 2.



If the application isn't able to tolerate duplicates and we have to make sure that the messages arrive only once to the subscribers, QoS level 2 is the appropriate choice. However, magic comes with a price: we must take into account that QoS level 2 has the highest overhead, compared to the other QoS levels.

Test your knowledge

1. MQTT runs on top of:
 1. The MQIP protocol.
 2. The TCP/IP protocol.
 3. The IoT protocol.
2. The data for an MQTT message is known as:
 1. Payload.
 2. Packet.
 3. Upload.
3. In MQTT version 3.1.1, the broker is named:
 1. MQTT agent.
 2. MQTT client.
 3. MQTT server.
4. QoS level 0 for MQTT means:
 1. Exactly once delivery
 2. At most once delivery
 3. At least once delivery
5. QoS level 1 for MQTT means:
 1. Exactly once delivery
 2. At most once delivery
 3. At least once delivery
6. QoS level 2 for MQTT means:
 1. Exactly once delivery
 2. At most once delivery
 3. At least once delivery

Summary

In this chapter, we started our journey toward understanding the MQTT protocol. We understood convenient scenarios for this protocol, the details of the publish/subscribe pattern and message filtering. We learned basic concepts related to MQTT and understood the different components: clients, brokers and connections.

We learned to install a Mosquitto broker or server on Linux, macOS and Windows. We used the default configuration to learn how everything works under the hoods while using Mosquitto. This way, it will be easier for us to use the client libraries in different programming languages to publish MQTT messages and subscribe to MQTT topic filters.

We learned best practices related to topics, single level and multi level wildcards. We studied in detail the different Quality of Service levels supported by MQTT and when it is convenient to use each of them. We analyzed their advantages and disadvantages.

Now that we understood how the MQTT basics work, we will learn how to secure an MQTT server and to follow best practices related to security, which is what we are going to discuss in the next chapter.

Chapter 2. Securing an MQTT Mosquitto Server

In this chapter, we will secure a Mosquitto server. We will work with digital certificates to encrypt all the data sent between the MQTT clients and the server. To secure the server, we will:

- Generate a private certificate authority to use TLS with Mosquitto
- Create a certificate for the Mosquitto server
- Configure TLS transport security in Mosquitto
- Test the MQTT TLS configuration with MQTT.fx
- Test the MQTT TLS configuration with MQTT-spy
- Create a certificate for each MQTT client
- Configure TLS client certificate authentication in Mosquitto
- Test the MQTT TLS client authentication with MQTT.fx
- Test the MQTT TLS configuration with MQTT-spy
- Force the TLS protocol version

Securing a Mosquitto server

Security for IoT, mobile, and web applications is an extremely important topic that deserves entire books dedicated to it. Each solution has its own security requirements, and it is very important to consider all of them when developing each component of the solution.

If we use MQTT to publish values that are neither confidential nor critical for other applications, our only concern may be to keep control over the maximum number of subscribers to topics to make sure that the messages are always available. This way, we can avoid the failure of MQTT in delivering messages to a huge number of subscribers.

However, most of the times, we won't be working on a solution that can share the data with the entire world without limitations and doesn't need to care about data confidentiality and integrity in addition to data availability. Imagine that we are working on a solution that includes a mobile app that allows users to control a huge drone. If the drone flies the wrong way, we can do harm to real people. We cannot allow any unknown publisher to be able to send messages to the topics that allow us to control the drone. We have to make sure that the right person is controlling the drone and that the commands sent as part of the messages cannot be altered by an intruder in the middle of the road.

Tip

The different levels of security come at a price, that is, there is always an additional overhead. Hence, we should always keep a balance to avoid overheads that can make the entire solution unfeasible and unusable. Whenever we add more security, we will require additional bandwidth and we will add a processing overhead to the clients and the server. We have to take into account that some cryptographic algorithms that work without problems in modern smartphones aren't suitable for IoT boards with a constrained processing power. Sometimes, the security requirements can force us to use specific hardware, such as more powerful IoT boards. We definitely have to consider security before purchasing all the hardware for our solution.

Another important thing that we must take into account is that many security levels require maintenance tasks, which might be unfeasible in certain cases or extremely difficult to achieve in other cases. For example, if we decide to use a certificate for each device that will become a client of the MQTT server, we will have to generate and distribute a certificate for each device. We have to access the filesystem for the device to copy new files to it. In case we have to invalidate a certificate, it would be necessary to provide a new certificate to the affected device. Consider a situation in which all the devices are distributed in different locations, which are difficult to access; we must have a mechanism to remotely access the device and be able to provide it with the new certificate. This task would also require security because we don't want anybody else to access the device's filesystem. Thus, things can become extremely complex once we start analyzing all the security requirements and the possible and necessary maintenance tasks.

Each MQTT server or broker implementation can provide specific security features. We will work with some of the features provided by Mosquitto out of the box. Specific security requirements might make us decide to work with a specific MQTT server or broker implementation.

When we work with Mosquitto, we can implement security in the following levels:

- **Network:** We can use a **Virtual Private Network (VPN)** to extend a private network across the Internet.
- **Transport:** MQTT uses TCP as the transport protocol, and therefore, by default, the communications aren't encrypted. **Transport Layer Security (TLS)** is usually referred to as TLS/SSL, because **Secure Socket Layers (SSL)** is its predecessor. We can use TLS to secure and encrypt communications between the MQTT clients and the MQTT server. The usage of TLS with MQTT is sometimes referred to as MQTTS. TLS allows us to provide both privacy and data integrity. We can use TLS client certificates to provide authentication.
- **Application:** At this level, we can take advantage of the features included in MQTT to provide application-level authentication and authorization. We can use `ClientId` (Client Identifier) to identify each client and combine it with user name and password authentication. We can add additional security mechanisms at this level. For example, we can encrypt the message payload and/or add integrity checks to ensure data integrity. However, the topic would still be unencrypted, and therefore, TLS is the only way of making sure that everything is encrypted. We can work with plugins to provide more complex authentication and authorization mechanisms. We can grant or deny permissions to each user to control which topics they can subscribe to and which topics they can publish messages to.

Tip

The most popular MQTT implementations provide support for TLS. However, make sure that you check the features before selecting the appropriate MQTT server for your solution.

We won't cover all the security topics because it would require one or more entire books dedicated to these topics. Instead, we will focus on the most common features used in transport-level security first, and then, we will move to application-level security. The usage of VPNs is out of the scope of this book. However, you must consider their usage based on your specific needs. We will use Mosquitto for our examples, but you can follow many similar procedures for any other MQTT server you decide to use. Everything we will learn will be useful for any other MQTT server that provides support for the same security features that we will use with Mosquitto.

Generating a private certificate authority to use TLS with Mosquitto

So far, we have been working with a Mosquitto server with its default configuration that listens on port 1883 and uses TCP as the transport protocol. The data sent between each MQTT client and server isn't encrypted. There are no restrictions to subscribers or publishers. If we open the firewall ports and redirect the ports in the router, any MQTT client that has our IP can publish to any topic and can subscribe to any topic.

In our examples in the previous chapter, we didn't make any changes in our configurations to allow incoming connections to port 1883, and therefore, we didn't open our Mosquitto server to the Internet.

We want to use TLS with MQTT and Mosquitto. This way, we will make sure that we can trust the MQTT server because we have confidence that it is who it says; our data will be private because it will be encrypted, and it will have integrity because it won't be altered in the middle of the road. In case you have experience with the `HTTP` protocol, we will make the same shift we do when we move from using `HTTP` to `HTTPS`.

Websites purchase certificates from major certificate authorities. In case we want to use a purchased certificate for the server, we don't need to generate our own certificates. In fact, it is the most convenient option when we have an MQTT server made public.

We will use the free OpenSSL utility to generate the necessary certificates for the server to enable TLS with Mosquitto. OpenSSL is already installed in macOS and most modern Linux distributions. In Windows, we already installed OpenSSL as one of the prerequisites for Mosquitto. The usage of the OpenSSL utility deserves an entire book, and therefore, we will just focus on generating the certificate we need with the most common options. In case you have any specific security needs, make sure that you dive deep into the necessary options to achieve the goals with OpenSSL.

Specifically, we will generate an X.509 digital certificate that uses the X.509 **Public Key Infrastructure (PKI)** standard. This digital certificate allows us to confirm that a specific public key belongs to the subject included within the certificate. There is an identity that issues the certificate and its details are also included in the certificate.

The digital certificate is valid only within a specific period, and therefore, we must take into account that a digital certificate will expire some day and we will have to provide a new certificate to replace the expired one. There are specific data requirements for the certificate based on the specific X.509 version that we use. According to the version and to the options we use to generate the certificates, we may need to provide specific data.

We will be running commands to generate different X.509 digital certificates and we will provide all the necessary details that will be included in the certificate. We will understand all the data that the certificate will have when we create them.

We will create our own private certificate authority, also known as CA. We will create a root certificate and then we will generate the server key.

Tip

Check the directory or folder in which you have installed OpenSSL. In macOS, OpenSSL is installed in `/usr/bin/openssl`. However, it is an old version and it is necessary to install a newer version before running the commands. It is possible to install the new version with the `homebrew` package manager. In Windows, the OpenSSL version we have installed as a prerequisite has the `openssl.exe` executable file in the default `C:\OpenSSL-Win32\bin` folder. Use the full path to OpenSSL in each of the next commands that start with `openssl`.

Create a new directory named `certificates` and change the necessary permissions for this directory to make sure that only you can access its contents.

Open the Terminal in macOS or Linux, or the Command Prompt in Windows, and go to the previously created directory, `certificates`. Run the following command to create a 2,048-bit root key and save it in the `ca.key` file: **`openssl genrsa -out ca.key 2048`**

The following lines show a sample output generated by the previous command: **Generating RSA private key, 2048 bit long modulus**

```
.....+  
.....++ e is 65537 (0x10001)
```

The previous command will generate the private root key in the `ca.key` file. Make sure that you keep this file private because anybody who has this file will be able to generate certificates. It is also possible to password-protect this file by using other options with `openssl`. However, as previously explained, we will follow the necessary steps to use TLS, and you can explore additional options related to OpenSSL and certificates.

Go to the Terminal in macOS or Linux, or the Command Prompt in Windows. Run the following command to self-sign the root certificate. The next command uses the previously created 2,048-bit private key saved in the `ca.key` file and generates a `ca.crt` file with the self-signed X.509 digital certificate. The command makes the self-signed certificate valid for 3,650 days. The value is specified after the `-days` option: **`openssl req -x509 -new -nodes -key ca.key -sha256 -days 3650 -out ca.crt`**

Tip

In this case, we specified the `-sha256` option to use SHA-256 hash functions. If we want increased security, we can use the `-sha512` instead of `-sha256` in all the cases in which we are using `-sha256`. This way, we will use SHA-512 hash functions. However, we must take into account that SHA-512 might not be appropriate for certain power-constrained IoT devices.

After you enter the previous command, OpenSSL asks information that will be incorporated in the certificate. You have to enter the information and press *Enter*. In case you don't want to enter specific information, just enter a dot (.) and press *Enter*. It is possible to pass all the values as arguments for the

openssl command, but it makes it a bit difficult to understand what we are doing. In fact, it is also possible to use fewer calls to the openssl command to perform the previous tasks. However, we will run a few more steps to understand what we are doing.

The following lines show the sample output and the questions with sample answers. Remember that we are generating our private certificate authority: **You are about to be asked to enter information that will be incorporated into your certificate request. What you are about to enter is what is called a Distinguished Name or a DN. There are quite a few fields but you can leave some blank For some fields there will be a default value, If you enter '.', the field will be left blank.** ----- Country Name (2 letter code) [AU]:US State or Province Name (full name) [Some-State]:CALIFORNIA Locality Name (eg, city) []:SAN FRANCISCO Organization Name (eg, company) [Internet Widgits Pty Ltd]:CERTIFICATE AUTHORITY Organizational Unit Name (eg, section) []: Common Name (e.g. server FQDN or YOUR name) []:CERTIFICATE AUTHORITY Email Address []:certificate@example.com

Run the following command to display the data and details for the generated certificate authority certificate file: **openssl x509 -in ca.crt -noout -text**

The following lines show the sample output, which displays details about the signature algorithm, the issuer, the validity, the subject, the extensions, and the constraints: **Certificate: Data: Version: 3 (0x2)** **Serial Number: ff:db:b4:d7:fa:30:00:bf** **Signature Algorithm: sha256WithRSAEncryption** **Issuer: C=US, ST=CALIFORNIA, L=SAN FRANCISCO, O=CERTIFICATE AUTHORITY, CN=CERTIFICATE AUTHORITY/emailAddress=certificate@example.com** **Validity Not Before: Jan 17 22:10:24 2017 GMT** **Not After : Jan 15 22:10:24 2027 GMT** **Subject: C=US, ST=CALIFORNIA, L=SAN FRANCISCO, O=CERTIFICATE AUTHORITY, CN=CERTIFICATE AUTHORITY/emailAddress=certificate@example.com** **Subject Public Key Info: Public Key Algorithm: rsaEncryption** **RSA Public Key: (2048 bit)** **Modulus (2048 bit):**
00:db:92:26:c1:fc:60:83:f1:c1:01:5e:46:e8:27: a8:e1:6e:08:02:65:05:ac:3e:d9:ad:22:8e:73:88:
3e:bf:16:b7:56:d7:44:87:e6:59:95:91:63:40:6a: 7c:fc:4f:db:48:b2:eb:d0:ca:ef:2d:6a:16:b9:31:
fc:8e:ce:e9:9b:e8:d7:1c:16:81:17:b8:93:c2:e2: 7c:14:0e:aa:4d:c4:75:38:4e:80:50:47:6d:fa:52:
c6:0c:98:2f:93:af:da:36:67:60:34:79:80:7e:cd: 03:da:c8:23:3d:e0:74:96:37:3c:0f:1d:fd:eb:e9:
b9:d2:4b:00:e8:a4:d9:f6:e1:d2:51:e7:e5:74:29: dc:3d:68:84:93:47:7a:cf:6e:92:e3:10:db:73:95:
7c:d4:c7:54:b0:10:bd:5c:0f:1e:0a:c1:bd:d9:25: d8:0d:d2:c2:30:92:90:6c:2d:4c:59:17:e0:d9:06:
7d:c2:2a:b7:80:b8:cf:09:7b:23:e9:ed:8e:75:78: b7:8f:9e:87:17:a6:17:2d:c0:68:86:0d:97:5e:ab:
5f:8d:d2:66:a7:11:0e:89:51:9c:9e:be:22:3f:f2: bd:ca:03:c5:da:5d:d0:09:20:9b:55:60:23:6b:3c:
67:de:99:8a:ef:0a:62:7f:92:8d:88:c9:01:70:b8: 3d:eb Exponent: 65537 (0x10001) X509v3
extensions: X509v3 Subject Key Identifier:

60:E5:C6:17:15:F8:2B:54:B2:DA:33:25:61:D4:4B:E3:DC:D9:70:6B X509v3 Authority Key Identifier: keyid:60:E5:C6:17:15:F8:2B:54:B2:DA:33:25:61:D4:4B:E3:DC:D9:70:6B
DirName:/C=US/ST=CALIFORNIA/L=SAN FRANCISCO/O=CERTIFICATE AUTHORITY/CN=CERTIFICATE AUTHORITY/emailAddress=certificate@example.com
serial:FF:DB:B4:D7:FA:30:00:BF X509v3 Basic Constraints: CA:TRUE Signature Algorithm: sha256WithRSAEncryption b8:b5:dc:71:b4:7f:66:f4:c0:d2:8f:0a:87:63:4f:b3:74:b0:
37:dd:5a:20:10:01:1c:91:ba:fa:71:47:b7:ad:ec:37:69:c1:
31:79:58:70:89:f2:d4:67:b3:a6:64:e2:75:0d:48:d1:fc:df:

```
67:3d:e7:cc:fa:9c:6d:65:ef:39:b2:74:76:11:4b:b5:23:3f:  
b3:cb:af:7a:eb:33:d0:4c:da:ed:1e:40:4b:33:7b:ed:25:ba:  
9a:5a:d9:45:36:54:b7:25:b2:96:37:32:be:b1:f5:fa:1b:0f:  
e4:29:7a:bd:c5:78:da:cc:ff:f0:3b:16:71:13:b0:ee:6c:04:  
79:b0:33:56:c6:ac:28:d7:0b:7e:0e:ca:f0:7d:56:0f:59:e8:  
5a:f9:49:66:1b:b8:7e:a1:31:07:ed:f6:0c:ec:9a:1e:82:73:  
26:c5:2b:a9:c5:44:ef:b3:a4:7d:98:23:3e:f0:5b:ba:af:b6:  
bf:0c:c8:be:75:c9:a6:07:2e:4d:47:2d:71:80:fc:fd:42:e2:  
18:0d:4d:65:7e:6f:27:42:65:79:83:f9:74:11:c6:70:32:6f:  
d9:98:5c:54:e5:eb:b6:68:7b:93:25:56:ef:11:d4:2f:61:d1:  
43:4f:b9:69:1c:2d:d2:e0:99:e5:12:be:20:c1:54:fe:b8:ea: c9:f0:27:d9
```

After running the previous commands, we will have the following two files in the `certificates` directory:

- `ca.key`: Certificate authority key
- `ca.crt`: Certificate authority certificate file

Tip

The certificate authority certificate file is in a **Privacy Enhanced Mail (PEM)** format. We must remember this format because some MQTT utilities will require us to specify whether the certificate is in the PEM format or not. A wrong value in this option won't make the MQTT client establish a connection with an MQTT server that uses a certificate in PEM format.

Creating a certificate for the Mosquitto server

Now that we have a private certificate authority, we can create the certificate for the Mosquitto server, that is, a certificate for the computer that will run the MQTT server.

First, we must generate a new private key that will be different from the private key we generated for our own private certificate authority.

Go to the Terminal in macOS or Linux, or the Command Prompt in Windows. Run the following command to create a 2,048-bit key and save it in the `server.key` file: **openssl genrsa -out server.key 2048**

The following lines show a sample output generated by the previous command: **Generating RSA private key, 2048 bit long modulus**

```
.....+  
.....++ e is 65537 (0x10001)
```

The previous command will generate the private key in the `server.key` file. Go back to the Terminal in macOS or Linux, or the Command Prompt in Windows. Run the following command to generate a certificate signing request, also known as CSR. The next command uses the previously created 2,048-bit private key saved in the `server.key` file and generates a `server.csr` file: **openssl req -new -key server.key -out server.csr**

After you enter the previous command, OpenSSL asks for information that will be incorporated in the certificate. You have to enter the information and press *Enter*. In case you don't want to enter specific information, just enter a dot (.) and press *Enter*. In this case, the most important value is **Common Name**. Enter the IPv4 or IPv6 address for the computer that is running the Mosquitto server in this field.

The following lines show the sample output and the questions with sample answers. Do not forget to enter the appropriate value for **Common Name**: **You are about to be asked to enter information that will be incorporated into your certificate request. What you are about to enter is what is called a Distinguished Name or a DN. There are quite a few fields but you can leave some blank For some fields there will be a default value, If you enter '.', the field will be left blank. ----- Country Name (2 letter code) [AU]:US State or Province Name (full name) [Some-State]:CALIFORNIA Locality Name (eg, city) []:PALO ALTO Organization Name (eg, company) [Internet Widgits Pty Ltd]:MQTT SERVER Organizational Unit Name (eg, section) []:MQTT Common Name (e.g. server FQDN or YOUR name) []:192.168.1.101 Email Address []:mqttserver@example.com Please enter the following 'extra' attributes to be sent with your certificate request A challenge password []: An optional company name []:Mosquitto**

Go to the Terminal in macOS or Linux, or the Command Prompt in Windows. Run the following command to sign the previously created certificate signing request, that is, the `server.csr` file. The next command also uses the self-signed X.509 digital certificate for the certificate authority and its private key that we generated before: the `ca.crt` and `ca.key` files. The command generates a `server.crt` file with the signed X.509 digital certificate for the Mosquitto server. The command makes the signed certificate valid for 3,650 days. The value is specified after the `-days` option: **openssl x509 -req -in server.csr -CA**

```
ca.crt -CAkey ca.key -CAcreateserial -out server.crt -days 3650 -sha256
```

Tip

As we did when we created the self-signed X.509 digital certificate for the certificate authority, we also specify the - sha256 option to use SHA-256 hash functions for the Mosquitto server certificate. You can use the - sha512 option instead of - sha256 in case you want to use SHA-512 hash functions for increased security.

The following lines show a sample output generated by the previous command. The values shown after the subject will be different in your configuration because you entered your own values when you generated the certificate signing request that was saved in the server.csr file:

Signature ok
subject=/C=US/ST=CALIFORNIA/L=PALO ALTO/O=MQTT
SERVER/OU=MQTT/CN=192.168.1.101/emailAddress=mqttserver@example.com Getting CA
Private Key

Run the following command to display the data and details for the generated server certificate file:
openssl x509 -in server.crt -noout -text

The following lines show the sample output that displays details about the signature algorithm, the issuer, the validity, the subject, and the signature algorithms: Certificate: Data: Version: 1 (0x0) Serial Number: 87:f1:cb:ae:43:c3:17:ee Signature Algorithm: sha256WithRSAEncryption Issuer: C=US, ST=CALIFORNIA, L=SAN FRANCISCO, O=CERTIFICATE AUTHORITY, CN=CERTIFICATE AUTHORITY/emailAddress=certificate@example.com Validity Not Before: Jan 18 14:29:48 2017 GMT Not After : Jan 16 14:29:48 2027 GMT Subject: C=US, ST=CALIFORNIA, L=PALO ALTO, O=MQTT SERVER, OU=MQTT, CN=192.168.1.101/emailAddress=mqttserver@example.com Subject Public Key Info: Public Key Algorithm: rsaEncryption RSA Public Key: (2048 bit) Modulus (2048 bit): 00:df:f6:e3:92:27:44:c0:93:56:70:e1:ec:97:c0: fc:24:df:73:a1:4b:d9:a5:d9:30:0c:63:06:8b:4c: 69:91:42:fc:41:a1:ba:13:b9:8b:0d:b1:89:80:48: 20:b7:61:c0:4b:be:92:66:c8:a5:f4:67:b3:7e:05: ed:59:1c:1e:a1:6f:db:3b:1b:16:c8:cd:93:d7:c8: 28:0d:20:73:12:9e:1d:ad:30:af:97:a7:4e:be:1b: 90:48:d6:8c:d4:04:1a:3f:a1:2e:4a:dc:b2:d8:32: f1:c9:83:8a:d7:15:a1:da:58:69:38:f3:0b:50:b0: f6:42:c2:8d:c3:32:4a:20:e7:f6:3d:f2:59:70:d9: da:13:59:aa:4f:1a:83:e8:e4:9a:82:ca:db:81:89: 4b:06:48:94:c2:8a:10:0b:7d:c8:d7:2e:c4:25:54: 60:b3:13:77:f2:99:dc:1c:26:d5:99:d2:24:57:3e: 6e:80:c5:e7:49:30:2a:01:d4:b0:19:ef:48:ec:f5: 0f:8b:af:52:f6:0c:6e:91:2d:d8:e5:5b:49:cc:9a: 79:d7:13:1e:b2:c2:5b:d2:3c:b7:71:f3:f6:9c:50: 25:97:da:0c:7a:24:4a:7e:0a:6a:02:5f:ba:2d:b3: 5d:0b:e6:e2:d7:0c:7d:31:c3:28:68:3c:02:31:0d: 8d:33 Exponent: 65537 (0x10001) Signature Algorithm: sha256WithRSAEncryption
72:3b:9b:e7:c0:17:66:a2:3b:52:b6:12:74:e5:1d:21:fd:fc:
b3:09:9c:b1:f2:97:15:5e:44:0e:3f:ee:c3:e4:61:00:4f:16:
99:58:19:72:3c:3d:54:df:f8:2c:6c:51:2c:7f:50:f7:4b:24:
2e:cc:2f:f6:ee:43:97:b0:d5:05:a9:89:18:ac:18:26:28:45:
3c:0f:6e:ae:54:6a:1f:da:71:34:2e:01:79:5b:3b:14:f0:cd:
70:11:4c:4d:e5:36:89:e0:b2:2f:64:b5:44:05:65:94:35:a4:
0f:21:e8:ff:07:27:89:e6:02:cf:f7:51:68:2c:e0:c2:c2:0c:
dd:26:56:70:bc:26:64:ab:7e:cc:c0:0c:0b:bf:da:91:22:05:
71:41:af:af:c7:47:7a:cb:cd:f3:49:bb:bb:ca:52:f5:56:

d3:75:33:c6:9f:fb:a8:35:87:3e:d5:13:37:b4:c5:d4:19:64:
ab:f2:5d:b7:06:d8:c8:54:95:cd:f2:02:74:d7:49:62:55:73:
b3:80:78:85:c2:6b:aa:68:96:95:4c:c4:3c:3e:20:ed:4a:8f:
84:34:51:28:99:01:d1:4d:21:4d:8f:97:7a:47:8c:b0:8e:4a:
59:f6:21:44:da:af:54:84:b9:ce:95:b9:21:35:14:a6:45:2f: e5:bc:37:84

After running the previous commands, we will have the following three files in the `certificates` directory:

- `server.key`: Server key
- `server.csr`: Server certificate signing request
- `server.crt`: Server certificate file

Tip

The server certificate file is in PEM format, as happened with the certificate authority certificate file.

Configuring TLS transport security in Mosquitto

Now, we will configure Mosquitto to use TLS transport security and work with encrypted communications with different clients. Note that we haven't generated certificates for the clients, and therefore, we won't use client certificates for authentication. This way, any client that has the `ca.crt` file will be able to establish a communication with the Mosquitto server.

Go to the Mosquitto installation directory and create a new subdirectory named `certificates`. In Windows, you will need administrator privileges to access the default installation folder.

Copy the following files from the `certificates` directory, in which we have saved the certificate authority certificate and the server certificate, to the `certificates` subdirectory we recently created within the Mosquitto installation directory:

- `ca.crt`
- `server.crt`
- `server.key`

In case you are running the Mosquitto server in a Terminal window in macOS or Linux, press `Ctrl + C` to stop it. In Windows, stop the appropriate service. If you are running the Mosquitto server in Linux, run the following command to stop the service:

```
sudo service mosquitto stop
```

Go to the Mosquitto installation directory and open the `mosquitto.conf` configuration file. By default, all the lines for this file are commented, that is, they start with a hash sign (#). The default value for each setting is indicated and commented. This way, we easily know all the default values. The settings are organized in different sections.

Tip

It is a good practice to make a backup copy of the exiting `mosquitto.conf` configuration file before making changes to it. Whenever we make changes to `mosquitto.conf` it is a good idea to be able to easily roll back to the previous configuration if something goes wrong.

In macOS or Linux, add the following lines at the end of the configuration file and make sure you replace `/usr/local/etc/mosquitto/certificates` with the full path to the `certificates` directory we created within the Mosquitto installation folder:

```
# MQTT over TLS
listener 8883
cafile /usr/local/etc/mosquitto/certificates/ca.crt
certfile /usr/local/etc/mosquitto/certificates/server.crt
keyfile /usr/local/etc/mosquitto/certificates/server.key
```

In Windows, add the following lines at the end of the configuration file and make sure you replace `C:\Program Files (x86)\mosquitto\certificates` with the full path to the `certificates` directory we created within the Mosquitto installation folder:

```
# MQTT over TLS
listener 8883
cafile C:\Program Files (x86)\mosquitto\certificates\ca.crt
certfile C:\Program Files (x86)\mosquitto\certificates\server.crt
keyfile C:\Program Files (x86)\mosquitto\certificates\server.key
```

We specified the 8883 value for the `listener` option to make Mosquitto listen for incoming network connections on port 8883. This port is the default port number for MQTT with TLS.

The `cafile` option specifies the full path for the file that provides a PEM-encoded certificate authority certificate file: `ca.crt`.

The `certfile` option specifies the full path for the file that provides a PEM-encoded server certificate: `server.crt`.

Finally, the `keyfile` option specifies the full path for the file that provides a PEM-encoded server keyfile: `server.key`.

Save the changes to the `mosquitto.conf` configuration file and launch Mosquitto again with the same mechanism we learned for Linux, macOS, and Windows in the previous chapter. The Mosquitto server will be listening on port 8883 instead of 1883.

We will use the `mosquitto_sub` command-line utility included in Mosquitto to try to generate a simple MQTT client that subscribes to a topic and prints all the messages it receives. We will use the default configuration that will try to establish a communication with the Mosquitto server by using the default 1883 port without a certificate authority certificate specified. Open the Terminal in macOS or Linux, or the Command Prompt in Windows, go to the directory in which Mosquitto is installed, and run the following command:

```
mosquitto_sub -V mqttv311 -t sensors/drone01/altitude -d
```

The `mosquitto_sub` utility will display the following error. The Mosquitto server is not accepting connection on port 1883 anymore:

Error: Connection refused

Run the following command that uses the `-p` option followed by the port number that we want to use: 8883. This way, we will try to connect to port 8883 instead of the default 1883 port:

```
mosquitto_sub -V mqttv311 -p 8883 -t sensors/drone01/altitude -d
```

The `mosquitto_sub` utility will display debug messages indicating that it is sending the CONNECT packet to the MQTT server. However, the connection will never be established because Mosquitto will display the following error. The Mosquitto server is not accepting connection on port 1883 anymore. Press `Ctrl+C` to stop the utility trying to connect.

The following command specifies the certificate authority certificate file after the `--cafile` option in addition to indicating that we want to use port 8883. You just have to replace `ca.crt` with the full path to

the ca.crt file you created in the certificates directory. For example, it might be C:certificatesca.crt in Windows or /Users/gaston/certificates/ca.crt in macOS or Linux. The mosquitto_sub utility will create an MQTT subscriber that will establish an encrypted connection with Mosquitto:

```
mosquitto_sub -V mqttv311 -p 8883 --cafile ca.crt -t  
sensors/drone01/altitude-d
```

With a similar syntax, we will use the mosquitto_pub command-line utility included in Mosquitto to generate a simple MQTT client that publishes a message to a topic with an encrypted connection. Open the Terminal in macOS or Linux, or the Command Prompt in Windows, go to the directory in which Mosquitto is installed, and run the following command:

```
mosquitto_pub -V mqttv311 -p 8883 --cafile ca.crt -t  
sensors/drone01/altitude -m "150 f" -d
```

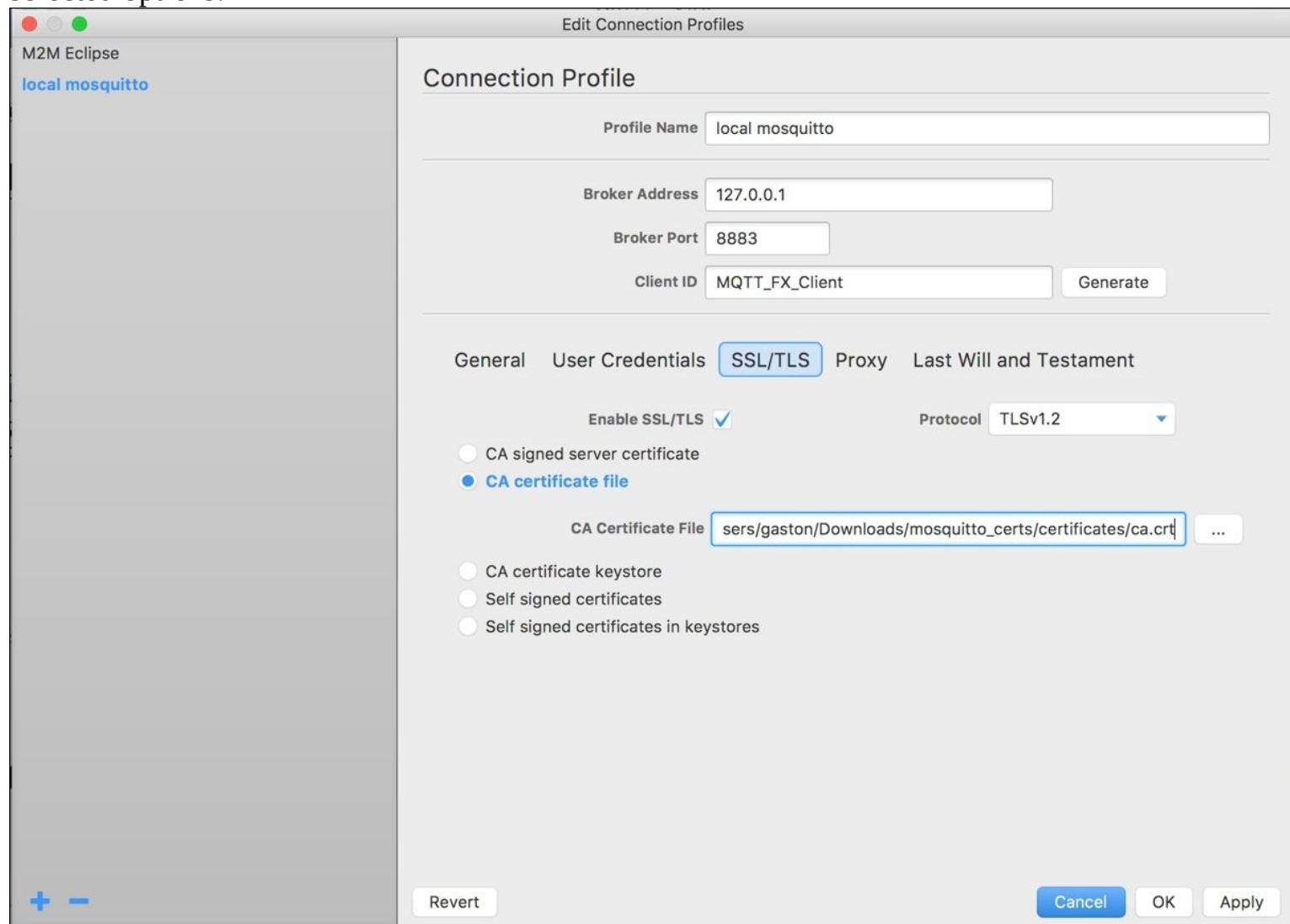
Tip

Remember to replace ca.crt with the full path to the ca.crt file you created in the certificates directory.

Testing the MQTT TLS configuration with MQTT.fx

Now, we will use the MQTT.fx GUI utility to generate another MQTT client that uses an encrypted connection to publish messages to the same topic, `sensors/drone01/altitude`. We have to make changes to the connection options to enable TLS and specify the certificate authority certificate file. Follow the next steps:

1. Launch MQTT.fx, select **local mosquitto** in the dropdown located on the upper-left corner, and click on the configuration icon on the right-hand side of this dropdown and on the left-hand side of the **Connect** button. MQTT.fx will display the **Edit Connection Profiles** dialog box with different options for the connection profile named **local mosquitto**.
2. Change the value specified in **Broker Port** from **1883** to **8883** .
3. Press the **SSL/TLS** button.
4. Activate the **Enable SSL/TLS** checkbox.
5. Activate the **CA certificate file** radio button.
6. Enter or select the full path to the `ca.crt` file that you created in the `certificates` folder in the **CA Certificate File** textbox and click **OK**. The following screenshot shows the dialog box with the selected options:



- Click on the **Connect** button. MQTT.fx will establish an encrypted connection with the local Mosquitto server. Notice that the **Connect** button is disabled and the **Disconnect** button is enabled because the client is connected to the Mosquitto server.
- Click **Subscribe** and enter `sensors/drone01/altitude` in the dropdown on the left-hand side of the **Subscribe** button. Then, click the **Subscribe** button. MQTT.fx will display a new panel on the left-hand side with the topic to which we have subscribed.
- Click **Publish** and enter `sensors/drone01/altitude` in the dropdown on the left-hand side of the **Publish** button.
- Enter the following text in the textbox below the **Publish** button: `400 f`.
- Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic.
- Click **Subscribe** and you will see the published message.

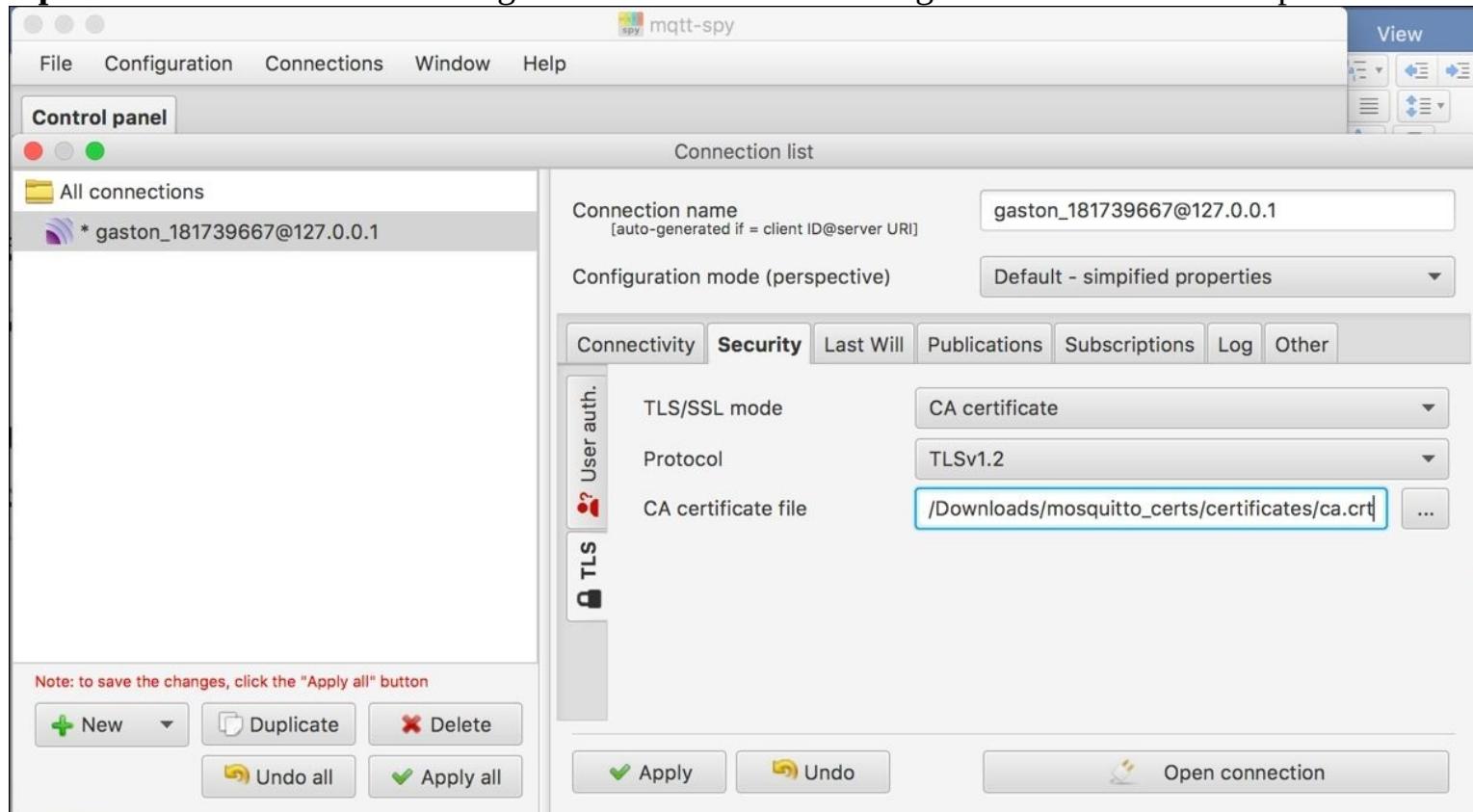
With the configuration changes we have made to the Mosquitto server, any client that has the certificate authority certificate file, that is, the `ca.crt` file we generated, will be able to establish a connection with Mosquitto, subscribe, and publish to topics. The data sent between the MQTT clients and the MQTT server is encrypted. In this configuration, we don't require the MQTT clients to provide certificates for authentication.

Testing the MQTT TLS configuration with MQTT-spy

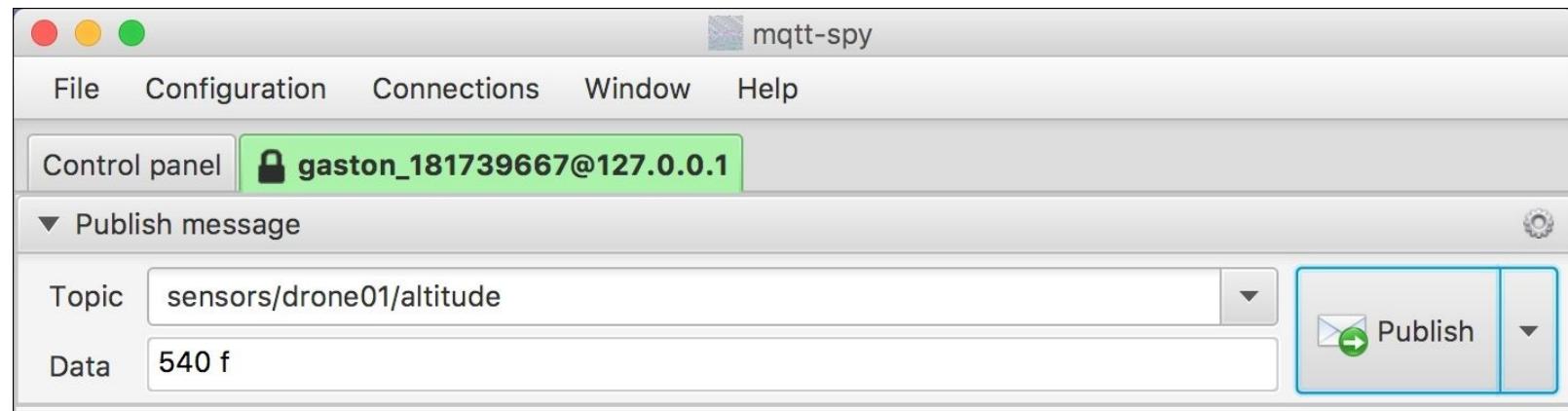
There is another very popular utility to generate MQTT clients that can subscribe to topics and publish to topics: MQTT-spy. This utility is open source and can run on any computer that has Java 8 or a higher version installed on it. You can find more information about MQTT-spy at <http://kamifb.github.io/mqtt-spy>. The options to establish a connection with an MQTT server with a certificate authority certificate file are similar to the ones we analyzed for MQTT.fx. However, in case you also want to work with this utility, it is convenient to analyze them in detail.

Now, we will use the MQTT-spy GUI utility to generate another MQTT client that uses an encrypted connection to publish messages to the same topic, that is, `sensors/drone01/altitude`. Follow the next steps:

1. Launch MQTT-spy.
2. Select **Connections | New connection**. The **Connection list** dialog box will appear.
3. Click on the **Connectivity** tab and select **MQTT 3.1.1** in the **Protocol version** dropdown. We want to work with MQTT version 3.1.1.
4. Click on the **Security** tab and on the **TLS** tab below the **User auth.** tab.
5. Select **CA certificate** in the **TLS/SSL mode** dropdown.
6. Select **TLSv1.2** in the **Protocol** dropdown.
7. Enter or select the full path to the `ca.crt` file that you created in the `certificates` folder and click **Open connection**. The following screenshot shows the dialog box with the selected options:



- MQTT-spy will close the dialog box and will display a new tab with a green background and the connection name that appears highlighted and selected on the left-hand side in the **Connection list** dialog. Make sure you click on the tab for the new connection.
- Enter `sensors/drone01/altitude` in the **Topic** dropdown.
- Enter the following text in the **Data** textbox: `540 f`. The following screenshot shows the tab for the new connection and the data entered in the different controls:



- Click the **Publish** button. MQTT-spy will publish the entered text to the specified topic, and you will be able to see the message in both the `MQTT.fx` subscriber and the `mosquitto-sub` subscriber.

Creating a certificate for each MQTT client

Now, we want to require each MQTT client to provide a valid certificate to establish a connection with the MQTT server. This way, only the clients that have a valid certificate will be able to publish or subscribe to topics. We will use the previously created private certificate authority to create client certificates for authentication.

We will generate a sample certificate for our local computer that will act as a client. We can follow the same procedure to generate additional certificates for additional devices that we want to be able to connect to the Mosquitto server. We just need to use a different name for the file and use a different device name in the corresponding option.

Tip

We must use the same certificate authority certificate that we used to generate the server certificate to generate the client certificates.

First, we must generate a new private key that will be different from the private keys we generated for our own private certificate authority and for the server certificate.

Go to the Terminal in macOS or Linux, or the Command Prompt in Windows. Run the following command to create a 2,048-bit key and save it in the `device001.key` file. To repeat this procedure for other devices, replace `device001` with any other name that identifies the device that will use the certificate. Do this in all the following commands that use `device001` for different file names and values: **openssl genrsa -out device001.key 2048**

The following lines show a sample output generated by the previous command: **Generating RSA private key, 2048 bit long modulus**

```
.....+++
.....+++ e is 65537 (0x10001)
```

The previous command will generate the private key in the `device001.key` file. Go back to the Terminal in macOS or Linux, or the Command Prompt in Windows. Run the following command to generate a certificate signing request, also known as CSR. The next command uses the previously created 2,048-bit private key saved in the `device001.key` file and generates a `device001.csr` file: **openssl req -new -key device001.key -out device001.csr**

After you enter the previous command, OpenSSL asks for information that will be incorporated in the certificate. You have to enter the information and press *Enter*. In case you don't want to enter specific information, just enter a dot (.) and press *Enter*. In this case, the most important value is **Common Name**. Enter the device name in this field: **You are about to be asked to enter information that will be incorporated into your certificate request. What you are about to enter is what is called a Distinguished Name or a DN. There are quite a few fields but you can leave some blank For some fields there will be a default value, If you enter '.', the field will be left blank. ----- Country Name (2 letter code) [AU]:US State or Province Name (full name) [Some-State]:CALIFORNIA Locality**

Name (eg, city) []:SAN JOSE Organization Name (eg, company) [Internet Widgits Pty Ltd]:MQTT CLIENT 001 Organizational Unit Name (eg, section) []:MQTT CLIENT 001 Common Name (e.g. server FQDN or YOUR name) []:MQTT CLIENT 001 Email Address []:mqttclient001@example.com Please enter the following 'extra' attributes to be sent with your certificate request A challenge password []: An optional company name []:.

Go to the Terminal in macOS or Linux, or the Command Prompt in Windows. Run the following command to sign the previously created certificate signing request, that is, the `client001.csr` file. The next command also uses the self-signed X.509 digital certificate for the certificate authority and its private key that we generated before: the `ca.crt` and `ca.key` files. The command generates a `client001.crt` file with the signed X.509 digital certificate for the MQTT client. The command makes the signed certificate valid for 3,650 days. The value is specified after the `-days` option. The `-addTrust clientAuth` option indicates that we want to use the certificate to authenticate a client: **openssl x509 -req -in device001.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out device001.crt -days 3650 -sha256 -addtrust clientAuth**

The following lines show a sample output generated by the previous command. The values shown after the subject will be different in your configuration because you entered your own values when you generated the certificate signing request that was saved in the `device001.csr` file: **Signature ok subject=/C=US/ST=CALIFORNIA/L=SAN JOSE/O=MQTT CLIENT 001/OU=MQTT CLIENT 001/CN=MQTT CLIENT 001/emailAddress=mqttclient001@example.com Getting CA Private Key**

Run the following command to display the data and details for the generated server certificate file: **openssl x509 -in device001.crt -noout -text**

The following lines show the sample output that displays details about the signature algorithm, the issuer, the validity, the subject, and the signature algorithms: **Certificate: Data: Version: 1 (0x0) Serial Number: 87:f1:cb:ae:43:c3:17:ef Signature Algorithm: sha256WithRSAEncryption Issuer: C=US, ST=CALIFORNIA, L=SAN FRANCISCO, O=CERTIFICATE AUTHORITY, CN=CERTIFICATE AUTHORITY/emailAddress=certificate@example.com Validity Not Before: Jan 18 22:51:42 2017 GMT Not After : Jan 16 22:51:42 2027 GMT Subject: C=US, ST=CALIFORNIA, L=SAN JOSE, O=MQTT CLIENT 001, OU=MQTT CLIENT 001, CN=MQTT CLIENT 001/emailAddress=mqttclient001@example.com Subject Public Key Info: Public Key Algorithm: rsaEncryption RSA Public Key: (2048 bit) Modulus (2048 bit): 00:a2:f0:b8:70:c6:9f:de:a0:42:27:fe:5c:86:2d: 73:a5:ae:1c:78:6c:bf:07:fe:bb:cc:a4:69:1d:3c:10:01:ef:4d:2c:85:43:57:4d:a2:5a:1c:ea:8f:1d: c2:73:4e:d8:27:31:78:9d:02:12:a9:2d:1b:2c:63:fc:9f:41:94:06:f6:7d:9b:4b:8f:58:cb:bf:b6:ea: 1d:e5:e2:fd:18:64:9a:f6:0e:ba:f0:7a:ba:52:c0:01:78:42:74:43:67:97:f4:db:80:a8:8b:e7:c0:c4: e7:ce:e4:ba:f8:7e:17:4f:e0:28:9d:93:ad:37:74:61:1b:fc:db:42:d1:36:31:11:47:9b:23:4f:37:d8: 54:e6:30:b7:f7:14:68:b7:e1:b3:43:43:f2:53:55: e6:ee:bf:d0:61:15:d3:7e:d1:de:88:1a:00:1b:b3: 09:bd:0f:ea:f8:d6:15:3b:8f:df:2c:3f:d1:ec:fa:3d:20:28:bb:82:ed:b2:3f:b7:db:c6:fb:99:16:70: 9a:2e:1c:3c:71:fd:ac:f0:ed:f1:ea:93:e6:2b:27:88:91:bb:32:1a:bf:90:31:bb:52:1f:22:60:9b:3a: 16:d3:02:98:97:8e:b1:49:1c:c8:d7:0f:61:24:bf:b1:ae:8c:fe:7d:99:6c:f4:d2:83:5e:63:77:4f:b4: e7:db Exponent: 65537 (0x10001) Signature Algorithm: sha256WithRSAEncryption 3c:85:23:6a:a5:cb:eb:94:59:58:53:fe:d0:c4:26:75:76:9f:fd:82:4b:5d:31:6a:a7:ba:08:75:96:f4:15:bf:e3:e2:7c:91:**

```
bf:50:19:e9:aa:76:99:13:fe:85:6d:22:67:96:68:b6:f0:86:  
06:cc:76:3c:79:4a:bc:1e:32:76:09:2e:82:93:b7:68:9c:1b:  
23:85:2c:ce:d2:39:d6:d3:c3:9c:d6:bc:38:25:bb:3c:55:53:  
a7:2f:58:d3:4d:9a:4c:86:78:7b:0f:6d:ea:8b:b3:0e:5c:2a:  
59:86:8e:1f:29:07:10:e8:37:ae:69:7e:b0:1f:a7:b9:08:e0:  
6e:aa:f9:a6:96:e0:e9:e4:4e:98:dc:a7:05:a1:ea:90:34:c9:  
4a:4d:7f:91:73:7a:22:24:91:82:37:d8:f4:a8:2a:1b:bb:b2:  
38:d5:27:ae:65:c5:e0:10:12:07:68:ca:ed:e9:27:1b:49:9b:  
39:81:bf:f2:6b:eb:b2:09:eb:1f:3b:de:dd:93:81:6f:c2:5a:  
4e:cd:b4:cd:05:fe:17:71:fb:fe:28:ba:55:4d:e4:91:64:fe:  
d6:57:61:81:d7:78:ae:9c:65:26:3c:6e:bb:9e:76:10:cd:94:  
44:92:1d:2d:e3:ff:57:fd:ab:24:b5:3c:cc:a8:76:0f:15:b3: 79:31:1f:ef
```

After running the previous commands, we will have the following three new files in the certificates directory:

- `device001.key`: Client key
- `device001.csr`: Client certificate signing request
- `device001.crt`: Client certificate file

Note

The client certificate file is in the PEM format, as happened with the authority certificate file and the server certificate file.

We will have to provide the following three files to any device that we want to be able to connect to the Mosquitto server:

- `ca.crt`
- `device001.crt`
- `device001.key`

Tip

Never provide additional files to the devices that have to establish a connection with the MQTT server. You don't want the devices to be able to generate additional certificates. You just want them to authenticate with a valid certificate.

The `openssl` utility allows us to provide values for many of the options with additional command-line options. Hence, it is possible to automate many of the previous steps to make it easier to generate multiple device certificates.

Configuring TLS client certificate authentication in Mosquitto

Now, we will configure Mosquitto to use TLS client certificate authentication. This way, any client will require the `ca.crt` file and a client certificate to establish a communication with the Mosquitto server.

In case you are running the Mosquitto server in a Terminal window in macOS or Linux, press `CtrlC` to stop it. In Windows, stop the appropriate service.

Go to the Mosquitto installation directory and open the `mosquitto.conf` configuration file.

In macOS, Linux, or Windows, add the following lines at the end of the configuration file:

```
require_certificate true
```

We specified the `true` value for the `require_certificate` option to make Mosquitto require a valid client certificate for any client that requests a connection to Mosquitto.

Save the changes to the `mosquitto.conf` configuration file and launch Mosquitto again. We will use the `mosquitto_sub` command-line utility included in Mosquitto to generate a simple MQTT client that subscribes to a topic filter and prints all the messages it receives. The following command specifies the certificate authority certificate file, the client certificate, and the client key. You have to replace `ca.crt`, `device001.crt`, and `device001.key` with the full path to these files created in the `certificates` directory. However, it is a better idea to copy these files to a new directory as if we were working with the files that will only be available to the device that wants to establish a connection with Mosquitto. The `mosquitto_sub` utility will create an MQTT subscriber that will establish an encrypted connection with Mosquitto and will provide the client certificate and client key to perform authentication.

```
mosquitto_sub -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t sensors/+/altitude -d
```

With a similar syntax, we will use the `mosquitto_pub` command-line utility included in Mosquitto to generate a simple MQTT client that publishes a message to a topic that will match the previously specified topic filter, with an encrypted connection and client authentication. Open the Terminal in macOS or Linux, or a Command Prompt in Windows, go to the directory in which Mosquitto is installed, and run the following command. Remember to replace `ca.crt`, `device001.crt`, and `device001.key` with the full path to these files created in the `certificates` directory:

```
mosquitto_pub -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t sensors/drone12/altitude -m "1350 f" -d
```

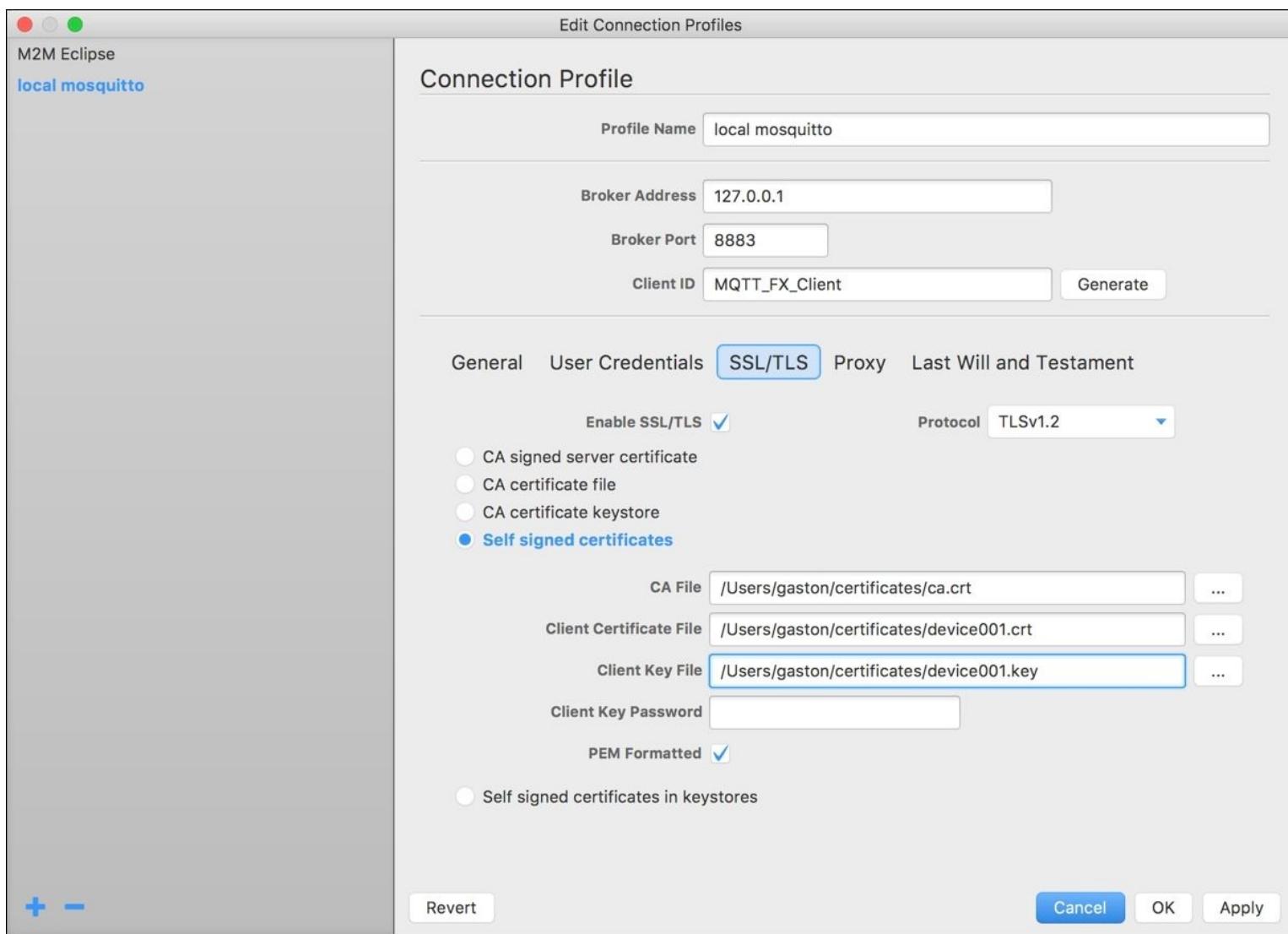
Tip

Sometimes, it is necessary to invalidate a client certificate. Mosquitto allows us to specify a PEM-encoded certificate revocation list file. We have to specify the path to this file as a value for the `crlfile` option in the Mosquitto configuration file.

Testing the MQTT TLS client authentication with MQTT.fx

Now, we will use the MQTT.fx GUI utility to generate another MQTT client that uses an encrypted connection and TLS client authentication to publish messages to a topic that matches the topic filter we used for the subscription, `sensors/drone25/altitude`. We have to make changes to the connection options we used when we enabled TLS and specify the client certificate and client key files. Follow the next steps:

1. Launch MQTT.fx, select **local mosquitto** in the dropdown located on the upper-left corner, and click on the configuration icon on the right-hand side of this dropdown and on the left-hand side of the **Connect** button. MQTT.fx will display the **Edit Connection Profiles** dialog box with different options for the connection profile named **local mosquitto**.
2. Make sure the value specified in **Broker Port** is **8883** .
3. Press the **SSL/TLS** button.
4. Make sure the **Enable SSL/TLS** checkbox is activated.
5. Activate the **Self signed certificates** radio button.
6. Enter or select the full path to the `ca.crt` file that you created in the `certificates` folder in the **CA File** textbox.
7. Enter or select the full path to the `device001.crt` file that you created in the `certificates` folder in the **Client Certificate File** textbox.
8. Enter or select the full path to the `device001.key` file that you created in the `certificates` folder in the **Client Key File** textbox.
9. Make sure that the **PEM Formatted** checkbox is activated. The following screenshot shows the dialog box with the selected options and sample values for the different textboxes:

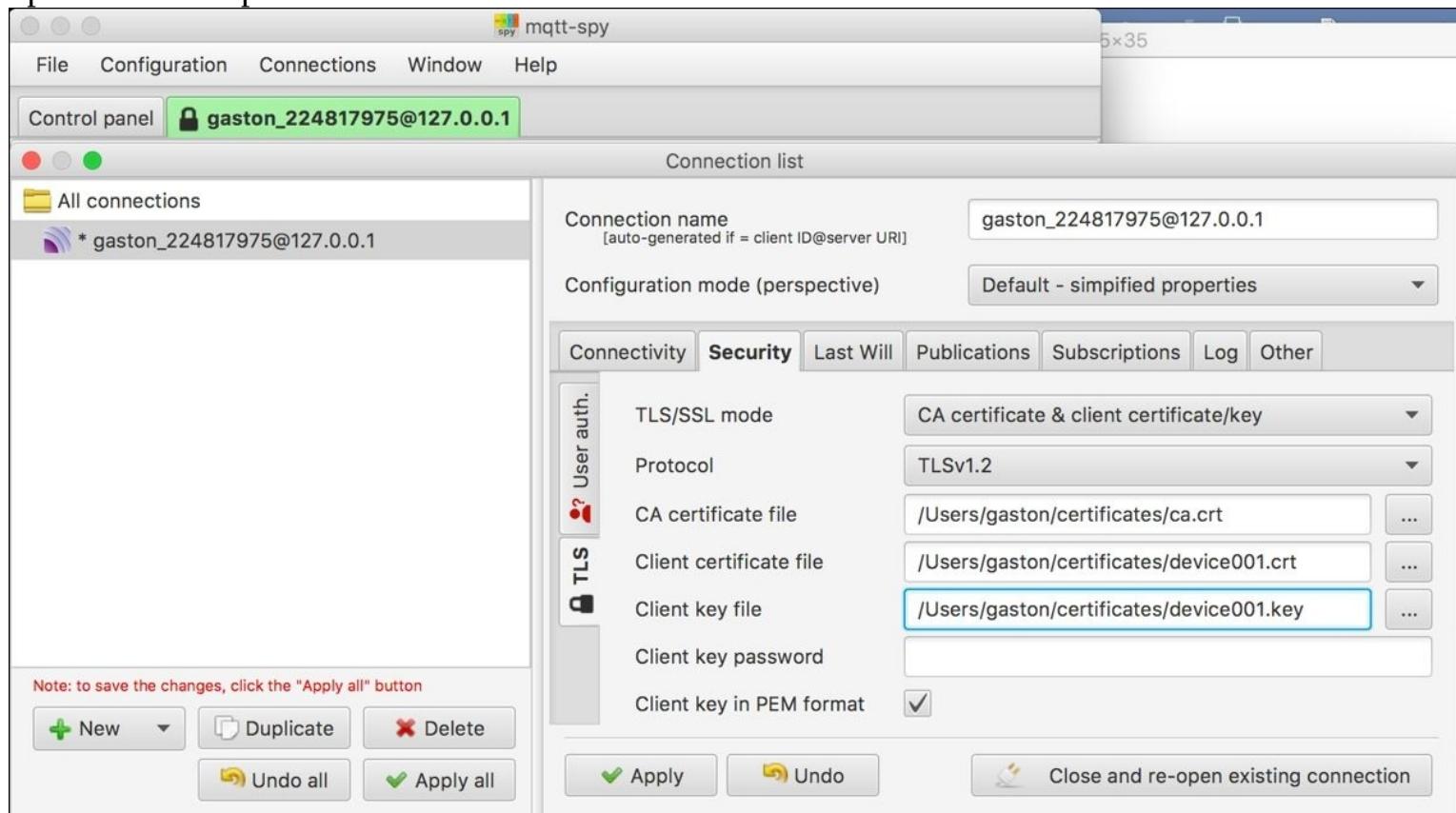


- Click **OK**. Then, click on the **Connect** button. MQTT.fx will establish an encrypted connection with the local Mosquitto server by using the certificate and key files we have specified. Note that the **Connect** button is disabled and the **Disconnect** button is enabled because the client is connected to the Mosquitto server.
- Click **Subscribe** and enter `sensors/+/altitude` in the dropdown on the left-hand side of the **Subscribe** button. Then, click the **Subscribe** button. MQTT.fx will display a new panel on the left-hand side with the topic filter to which we have subscribed.
- Click **Publish** and enter `sensors/drone40/altitude` in the dropdown on the left-hand side of the **Publish** button.
- Enter the following text in the textbox below the **Publish** button: `1235 f.`
- Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic.
- Click **Subscribe** and you will see the published message.

Testing the MQTT TLS configuration with MQTT-spy

Now, we will use the MQTT-spy GUI utility to generate another MQTT client that uses an encrypted connection to publish messages to another topic that will match the `sensors/+/altitude` topic filter, `sensors/drone500/altitude`. Follow the next steps:

1. Launch MQTT-spy.
2. Select **Connections | New connection** or **Connections | Manage** connections in case you were already running MQTT-spy or have saved previous settings. The **Connection list** dialog box will appear.
3. Click on the **Connectivity** tab and make sure **MQTT 3.1.1** is selected in the **Protocol version** dropdown.
4. Click on the **Security** tab and on the **TLS** tab below the **User auth.** tab.
5. Select **CA certificate & client certificate/key** in the **TLS/SSL mode** dropdown.
6. Select **TLSv1.2** in the **Protocol** dropdown.
7. Enter or select the full path to the `ca.crt` file that you created in the certificates folder in the **CA certificate file** textbox.
8. Enter or select the full path to the `device001.crt` file that you created in the certificates folder in the **Client certificate file** textbox.
9. Enter or select the full path to the `device001.key` file that you created in the certificates folder in the **Clientkey file** textbox.
10. Activate the **Client key in PEM format** checkbox. Finally, click **Open connection** or **Close and re-open existing connection**. The following screenshot shows the dialog box with the selected options and sample values for the textboxes:



- MQTT-spy will close the dialog box and will display a new tab with a green background and the connection name, which appears highlighted and selected on the left-hand side in the **Connection list** dialog. Make sure you click on the tab for the new connection.
- Enter sensors/drone500/altitude in the **Topic** dropdown.
- Enter the following text in the **Data** textbox: 1777 f.
- Click the **Publish** button. MQTT-spy will publish the entered text to the specified topic, and you will be able to see the message in both the MQTT.fx subscriber and the mosquitto-sub subscriber.

Tip

As happens with any security configuration, a checkbox that isn't activated based on the previous instructions won't allow the MQTT client to establish a connection with Mosquitto. It is very important to remember that the certificates use the PEM format.

Forcing the TLS protocol version

It is a good practice to use the highest possible TLS protocol version. By default, Mosquitto accepts TLS 1.0, 1.1, and 1.2. If all the clients are capable of working with the highest TLS protocol version supported by Mosquitto, we should force Mosquitto to use only the highest version. This way, we make sure that we won't be vulnerable to attacks to the previous TLS versions.

Now, we will make the necessary changes in the configuration file to force the usage of TLS 1.2. In case you are running the Mosquitto server in a Terminal window in macOS or Linux, press *Ctrl+C* to stop it. In Windows, stop the appropriate service.

Go to the Mosquitto installation directory and open the `mosquitto.conf` configuration file.

In macOS, Linux, or Windows, add the following lines at the end of the configuration file:

```
tls_version tlsv1.2
```

We specified the `tlsv1.2` value for the `tls_version` option to make Mosquitto work only with TLS 1.2. Any client that uses an earlier TLS version won't be able to establish a connection with the Mosquitto server.

Save the changes to the `mosquitto.conf` configuration file and launch Mosquitto again. We specified the TLS version when we configured the connection in both the MQTT.fx and MQTT-spy GUI utilities, and we specified TLS 1.2 as the desired TLS version for the client. We must use the `--tls-version tlsv1.2` option for the `mosquitto_sub` and `mosquitto_pub` command-line utilities.

Open the Terminal in macOS or Linux, or the Command Prompt in Windows, go to the directory in which Mosquitto is installed, and run the following command. Remember to use the full paths for the `ca.crt`, `device.001` and `device.key` files:

```
mosquitto_pub --tls-version tlsv1.2 -V mqttv311 -p 8883 --cafile ca.crt -t sensors/drone01/altitude -m "2532 f" -d
```

The previous command specified the usage of TLS 1.2, and therefore, the MQTT client could establish a connection with the Mosquitto server and publish the message. If we specify a different TLS version, the `mosquitto_pub` command won't be able to connect with the Mosquitto server.

Test your knowledge

1. Which of the following can we use to secure and encrypt communications between the MQTT clients and the MQTT server?
 1. TLS.
 2. TCP.
 3. HTTPS.
2. Which is the default port number for MQTT over TLS?
 1. 1883.
 2. 9883.
 3. 8883.
3. If we specify `true` as the value for the `require_certificate` option of the Mosquitto configuration file:
 1. Clients that want to connect to the MQTT server will require a client certificate.
 2. Clients that want to connect to the MQTT server won't require a client certificate.
 3. Clients that want to connect to the MQTT server can provide an optional client certificate.
4. Which of the following utilities allows us to generate an X.509 digital certificate?
 1. OpenX509.
 2. OpenSSL.
 3. TLS4You.
5. When we use MQTT over TLS:
 1. There is just a small bandwidth overhead but no processing overhead at all compared with MQTT over TCP without TLS.
 2. There are both bandwidth and processing overheads compared with MQTT over TCP without TLS.
 3. There is no overhead compared with MQTT over TCP without TLS.

Summary

In this chapter, we generated a private certificate authority, a server certificate, and client certificates to enable TLS transport security and TLS client authentication with Mosquitto. Communications between the MQTT clients and the MQTT server are encrypted.

We worked with OpenSSL to generate the digital certificates. We tested the MQTT TLS configuration with MQTT.fx, MQTT-spy, and the Mosquitto command-line utilities. We forced Mosquitto to use only a specific TLS version.

There are many other security topics related to MQTT servers and Mosquitto. We will work with some of them in the forthcoming chapters, in which we will develop applications that will use MQTT in different programming languages.

Now that you have understood how to encrypt communications between the MQTT clients and the Mosquitto server, we will control a drone with Python and MQTT, which is what we are going to discuss in the next chapter.

Chapter 3. Controlling a Drone with Python

In this chapter, we will write Python code to control a drone with MQTT messages. We will write code that will be able to run on different popular IoT platforms. We will:

- Understand the requirements to control a drone with MQTT
- Define the topics and commands to control a drone
- Install paho-mqtt for Python
- Connect the client to the MQTT server with Python
- Understand callbacks
- Subscribe to topics with Python
- Create a class to represent a drone
- Receive messages with Python
- Process commands with Python
- Work with the network loop with Python
- Send messages with Python

Understanding the requirements to control a drone with MQTT

In the previous chapters, we learned how MQTT works in detail. We understood how to establish a connection between an MQTT client and an MQTT server. You learned what happens when we subscribe to topic filters and when a publisher sends messages to specific topics. We installed and secured a Mosquitto server.

Now, we will use Python as our main programming language to generate MQTT clients that will act as publishers and subscribers. We will connect a Python MQTT client to the MQTT server and we will process commands to control a drone with MQTT messages.

We will use TLS encryption and TLS authentication because we don't want any MQTT client to be able to send commands to our drone. We want our Python code to run on many platforms because we will use the same code base to control drones that use the following IoT boards:

- Raspberry Pi 3
- Raspberry Pi 2 Model B
- Intel Edison
- Intel Galileo Gen 2
- Intel Joule 570x
- Intel Joule 550x

Depending on the platform, each drone is going to provide additional features because some boards are more powerful compared with other boards. However, we will focus on the basic features to keep our example simple and to stay concentrated on MQTT. Then, we will be able to use this project as a baseline for other solutions that require us to run code on an IoT board that runs Python code, has to connect to an MQTT server, and has to process commands.

The code that runs on the boards that power the drones must be able to process commands received in messages in a specific topic. We will use JSON strings in the payloads.

A client application also written in Python must be able to control one or more drones. We will also write the client application in Python and it will publish MQTT messages with JSON strings to the topic for each drone. The client application must display the results of executing each command. Each drone must publish a message to a specific topic whenever a command is successfully executed.

Defining the topics and commands

We will use the following topic name to publish the commands for a drone: `commands/dronename`, where `dronename` must be replaced with a unique name assigned to a drone. For example, if we assign `drone01` as the name for a drone that is powered by an Intel Joule 570x board, we will have to publish commands to the `commands/drone01` topic and the code that runs on this board will subscribe to this topic to receive messages with commands and react to them.

We will use the following topic name to make the drones publish details about the successfully executed commands: `processedcommands/dronename`, where `dronename` must be replaced with a unique name assigned to a drone. For example, if we assign `dronepi03` as the name for a drone that is powered by a Raspberry Pi 3 board, the client that wants to receive information about successfully processed commands has to subscribe to the `processedcommands/dronepi03` topic.

The commands will be sent in a JSON string with a key-value pair. The key must be equal to `COMMAND` and the value must specify any of the following valid commands. When the command requires additional parameters, the parameter name must be included in the next key and the value for this parameter in the value for this key.

- `TAKE_OFF`: Take off the drone.
- `LAND`: Land the drone.
- `LAND_IN_SAFE_PLACE`: Land the drone in a safe place that is configured for the drone.
- `MOVE_UP`: Make the drone move up.
- `MOVE_DOWN`: Make the drone move down.
- `MOVE_FORWARD`: Make the drone move forward.
- `MOVE_BACK`: Make the drone move back.
- `MOVE_LEFT`: Make the drone move left.
- `MOVE_RIGHT`: Make the drone move right.
- `ROTATE_RIGHT`: Make the drone rotate to the right. We must specify the degrees we want the drone to rotate right in the value for the `DEGREES` key.
- `ROTATE_LEFT`: Make the drone rotate to the left. We must specify the degrees we want the drone to rotate left in the value for the `DEGREES` key.
- `SET_MAX_ALTITUDE`: Set the maximum altitude that we allow for the drone. We must specify the desired maximum altitude in feet in the value for the `FEET` key.
- `SET_MIN_ALTITUDE`: Set the minimum altitude that we allow for the drone. We must specify the desired minimum altitude in feet in the value for the `FEET` key.

The following line shows an example of the payload for the command that takes off the drone:

```
{"COMMAND": "TAKE_OFF"}
```

The following line shows an example of the payload for the command that sets the maximum altitude for the drone to 20 feet:

```
{"COMMAND": "SET_MAX_ALTITUDE", "FEET": 20}
```

We have all the necessary details to start coding with Python.

Installing paho-mqtt for Python

The Eclipse Paho project provides an open source client implementation of MQTT. The project includes a Python client, also known as the Paho Python Client or Eclipse Paho MQTT Python client library. This Python client has been contributed from the Mosquitto project and was originally known as the Mosquitto Python Client.

The following is the web page for the Eclipse Paho project:<http://www.eclipse.org/paho>.

The following is the web page for the Eclipse Paho MQTT Python client library version 1.2, that is, the paho-mqtt module version 1.2: <https://pypi.python.org/pypi/paho-mqtt/1.2>.

Tip

We can use paho-mqtt in many modern IoT boards that support either Python 2.7.x or Python 3.x. For example, we can install paho-mqtt on Raspberry Pi boards, such as Raspberry Pi 3, and on Intel IoT boards, such as the Intel Joule, Intel Edison, and Intel Galileo. We just need to make sure that pip is installed to make it easier to install paho-mqtt. You can use your development computer to run the examples or any of the mentioned boards. You will need Python 2.7.x or Python 3.x installed on your development computer.

It is good practice to create a Python virtual environment and activate it before installing packages. This way, you can isolate the environment in which you are developing the sample application. To keep the example simple, we won't include all the instructions to create virtual environments in Python. However, in case you want to work with a virtual environment, make sure that you activate it before moving forward with the next steps.

In case you are going to work with an IoT board to run the example, make sure that you run all the commands in the SSH terminal or the Terminal window that runs on the board. In case you use your development computer, run the commands on a Terminal in macOS or Linux, or a Command Prompt in Windows. Make sure that you have activated your virtual environment in case you decided to work with one.

Now, we will use the pip installer to install paho-mqtt 1.2. We just need to run the following command in the SSH terminal or the local terminal window we are using with the board, or in the computer we are using to install the package:

```
pip install paho-mqtt
```

Tip

In case you are working with Intel boards, take into account that you must install pip before running the previous commands. On Raspberry Pi 3 boards, pip is already installed. If you are using your computer, the Python installation usually includes pip.

In case you have Python installed in the default folder in Windows and you aren't working with a Python

virtual environment, you will have to run the previous command in an Administrator Command Prompt.

In case you aren't working with a Python virtual environment in Raspbian (Raspberry Pi default operating system), you will have to run the following command:

```
sudo pip install paho-mqtt
```

The last lines for the output will indicate that the paho-mqtt package has been successfully installed. The output will be similar to the next lines but not exactly the same because it will vary based on the platform in which you are running the command:

```
Collecting paho-mqtt    Downloading paho-mqtt-1.2.tar.gz (41kB)    100%
|#####| 45kB 147kB/s Installing collected packages:
paho-mqtt    Running setup.py install for paho-mqtt Successfully installed paho-mqtt-1.2
```

Connecting the client to the MQTT server with Python

First, we will use `paho_mqtt` to create an MQTT client that connects to the Mosquitto MQTT server. We will write a few lines of Python code to establish a secured connection and subscribe to a topic.

In the previous chapter, we secured our Mosquitto server, and therefore, we will use the digital certificates we had created to authenticate the client. Most of the time, we will work with an MQTT server that uses TLS, and therefore, it is a good idea to learn how to establish a connection with TLS and TLS authentication. It is easier to establish an unsecured connection with an MQTT server, but it won't be the most common scenario we will face when developing applications that work with MQTT.

First, we need to copy the following files we had created in the previous chapter to a directory in the computer or device we will use to run a Python script. We saved the files in a directory called `certificates`. Create a `python_certificates` directory in the computer or board you are going to use as the MQTT client for this example. Copy the following three files to this new directory:

- `ca.crt`: Certificate authority certificate file
- `device001.crt`: Client certificate file
- `device001.key`: Client key

The following lines show a sample code that establishes a connection with an MQTT server, subscribes to the `test/drone01` topic filter, and prints all the messages received in the subscribed topic filter. You have to replace the value, `/Users/gaston/certificates`, in the `certificates_path` string with the path to the `python_certificates` directory you created. In addition, replace the value for `mqtt_server_host` with the IP address for the Mosquitto server or any other MQTT server you might decide to use. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_03` folder, in the `mqtt_essentials_example03_01.py` file:

```
import paho.mqtt.client as mqtt
import os.path

# Replace /Users/gaston/certificates with the path
# in which you saved the certificate authority file,
# the client certificate file and the client key
certificates_path = "/Users/gaston/certificates"
ca_certificate = os.path.join(certificates_path, "ca.crt")
client_certificate = os.path.join(certificates_path, "device001.crt")
# Replace localhost with the IP for the Mosquitto
# or other MQTT server
mqtt_server_host = "localhost"
mqtt_server_port = 8883
mqtt_keepalive = 60

def on_connect(client, userdata, rc):
```

```

print("Connect result: {}".format(mqtt.connack_string(rc)))
client.subscribe("test/drone01")

def on_subscribe(client, userdata, mid, granted_qos):
    print("Subscribed with QoS: {}".format(granted_qos[0]))

def on_message(client, userdata, msg):
    payload_string = msg.payload.decode('utf-8')
    print("Topic: {}. Payload: {}".format(
        msg.topic,
        str(msg.payload)payload_string))

if __name__ == "__main__":
    client = mqtt.Client(protocol=mqtt.MQTTv311)
    client.on_connect = on_connect
    client.on_subscribe = on_subscribe
    client.on_message = on_message
    client.tls_set(ca_certs = ca_certificate,
                   certfile=client_certificate,
                   keyfile=client_key)
    client.connect_async(host=mqtt_server_host,
                         port=mqtt_server_port,
                         keepalive=mqtt_keepalive)
    client.loop_forever()

```

The previous code uses the recently installed paho-mqtt module to establish an encrypted connection with the MQTT server, subscribe to the test/drone01 topic filter, and run the code when we receive messages in the topic. We will use this code to understand the basics of paho-mqtt.

The first line imports paho.mqtt.client as mqtt. This way, whenever we use the mqtt alias, we will be referencing paho.mqtt.client. Then, the code declares the certificates_path variable initialized with a string, that is, with the path in which you saved the certificate authority file, the client certificate file, and the client key. The code declares the following string variables with the full path to the certificate and key files we need to configure TLS and TLS client authentication: ca_certificate, client_certificate, and client_key. The call to os.path.join makes it easy to join the path specified in the certificates_path variable with the file name and generate the full path. The os.path.join function works for any platform, and therefore, we don't have to worry about whether to use a slash (/) or a backslash () to join the path with the file name. Sometimes, we can develop and test in Windows and then run the code on an IoT board that can use different Unix flavors. The usage of os.path.join makes our job easier in these scenarios in which we switch between different platforms.

The mqtt_server_host, mqtt_server_port and mqtt_keepalive variables specify the IP address for the MQTT server (the Mosquitto server), the port that we want to use (8883) and the number of seconds for the keep_alive option. It is very important to replace localhost with the IP address for the MQTT server. We specify 8883 for mqtt_server_port because we use TLS and this is the default port for MQTT over TLS, as we learned in the previous chapter.

Understanding callbacks

When we declare a function, we pass this function as an argument to another function or method, or we assign this function to an attribute, and then some code calls this function at some time; this mechanism is known as **callback**. The name callback is used because the code calls back a function at some time.

The code declares three functions that we will specify as callbacks:

- `on_connect`: This function will be called when the MQTT client receives a `CONNACK` response from the MQTT server, that is, when a connection has been successfully established with the MQTT server.
- `on_subscribe`: This function will be called when the MQTT client receives a `SUBACK` response from the MQTT server, that is, when a subscription has been successfully completed.
- `on_message`: This function will be called when the MQTT client receives a `PUBLISH` message from the MQTT server. Whenever the MQTT server publishes a message, based on the subscriptions for the client, this function will be called.

The code for the main block creates an instance of the `mqtt.Client` class (`paho.mqtt.client.Client`) that represents an MQTT client and that we will use to communicate with an MQTT broker. If we use the default parameters, we would work with MQTT version 3.1. We want to work with MQTT version 3.1.1, and therefore, we specified `MQTTv311` as the value for the `protocol` argument.

Then, the code assigns the `on_connect` function to the `client.on_connect` attribute, the `on_message` function to the `client.on_message` attribute, and the `on_subscribe` function to the `client.on_subscribe` attribute.

The call to the `client.tls_set` method configures encryption and authentication options. It is very important to call this method before running the `client.connect` method. We specify the full string paths to the certificate authority certificate file, the client certificate, and the client key in the `ca_certs`, `certfile` and `keyfile` arguments. The `ca_certs` argument name is a bit confusing, but we just need to specify the string path to the certificate authority certificate file, not multiple certificates.

Finally, the main block calls the `client.connect_async` method and specifies the values for the `host`, `port`, and `keepalive` arguments. This way, the code asks the MQTT client to establish a connection to the specified MQTT server.

The `connect_async` method runs with an asynchronous execution, and therefore, it is a non-blocking call. The equivalent method with a synchronous execution is the `connect` method. After a connection has been successfully established with the MQTT server, the specified callback in the `client.on_connect` attribute will be executed, that is, the `on_connect` function. This function receives the `mqtt.Client` instance that established the connection with the MQTT server in the `client` argument.

Tip

If you want to establish a connection with an MQTT server that isn't using TLS, you don't need to call the

`client.tls_set` method and use the appropriate port; by default, it will be 1883.

Subscribing to topics with Python

The code calls the `client.subscribe` method with "test/drone01" as an argument to subscribe to this specific single topic and the `qos` argument set to 2 to request a QoS level of 2.

Tip

In this case, we will only subscribe to one topic. However, it is very important to know that we are not limited to subscribe to a single topic filter; we may subscribe to many topic filters with a single call to the `subscribe` method.

After the MQTT server confirms the successful subscription to the specified topic filter, the specified callback in the `client.on_subscribe` attribute will be executed, that is, the `on_subscribe` function. This function receives a list of integers in the `granted_qos` argument that provides the QoS level that the MQTT server has granted for each of the topic filter subscription requests. The code in the `on_subscribe` function displays the QoS level granted by the MQTT server for the topic filter we specified.

Whenever there is a new message received that matches the topic filter to which we have subscribed, the specified callback in the `client.on_message` attribute will be executed, that is, the `on_message` function. This function receives the `mqtt.Client` instance that established the connection with the MQTT server in the `client` argument and an `mqtt.MQTTMessage` instance in the `msg` argument. The `mqtt.MQTTMessage` class describes an incoming message.

In this case, whenever the `on_message` function is executed, the value in `msg.topic` will always match "test/drone01" because we just subscribed to a single topic and no other topic name will match the topic filter. However, if we had subscribed to a topic filter or many topic filters with which more than one topic may match, it would be always necessary to check the topic in which the message was sent and in which we are receiving the message.

The code in the `on_message` function prints the topic for the message that has been received, `msg.topic`, and the string representation of the payload for the message, that is, the `msg.payload` attribute decoded to an UTF-8 string and saved in the `payload_string` variable.

Finally, the main block calls the `client.loop_forever` method that calls the `loop` method for us in an infinite blocking loop. At this point, we only want to run the MQTT client loop in our program. We will receive the messages whose topic matches the topic to which we have subscribed.

Make sure that the Mosquitto server or any other MQTT server you might want to use for this example is running. Then, execute the following line to start the example in any computer or device that you want to use as the MQTT client. Note that you might need to use `python3` instead of `python`, based on your configuration:

```
python mqtt_essentials_example03_01.py
```

In case you see a traceback with `SSLError` similar to the following lines, it means that the MQTT server

hostname or IP doesn't match the value specified for the Common Name attribute when you generated the server certificate file named `server.crt`. Make sure that you check the IP address for the MQTT server (Mosquitto server) and generate the server certificate file and key again with the appropriate IP address specified for Common Name, as explained in the previous chapter, in case you are working with the self-signed certificates we generated in the previous chapter:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Python/2.7/site-packages/paho/mqtt/client.py", line 612, in
connect
    return self.reconnect()
  File "/Library/Python/2.7/site-packages/paho/mqtt/client.py", line 751, in
reconnect
    self._tls_match_hostname()
  File "/Library/Python/2.7/site-packages/paho/mqtt/client.py", line 2331, in
_tls_match_hostname
    raise ssl.SSLError('Certificate subject does not match remote hostname.')
```

Now, follow these steps to use the MQTT.fx GUI utility to publish two messages to the `test/drone01` topic:

1. Launch MQTT.fx and establish a connection with the MQTT server by following the steps you learned in the previous chapter.
2. Click **Publish** and enter `test/drone01` in the dropdown on the left-hand side of the **Publish** button.
3. Click **QoS 2** on the right-hand side of the **Publish** button.
4. Enter the following text in the textbox below the **Publish** button: `{"COMMAND": "TAKE_OFF"}`. Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic.
5. Enter the following text in the textbox below the **Publish** button: `{"COMMAND": "LAND"}`. Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic.

In case you don't want to work with the MQTT.fx utility, you can run two `mosquitto_pub` commands to generate MQTT clients that publish messages to the topic. You just need to open another Terminal in macOS or Linux, or another Command Prompt in Windows, go to the directory in which Mosquitto is installed, and run the following commands. In this case, it isn't necessary to specify the `-d` option. Replace `localhost` with the IP for the MQTT server. Remember to replace `ca.crt`, `device001.crt`, and `device001.key` with the full path to these files created in the `certificates` directory:

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt
--key device001.key -t test/drone01 -m '{"COMMAND": "TAKE_OFF"}' -q 2 --tls-version
tlsv1.2-d
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt
--key device001.key -t test/drone01 -m '{"COMMAND": "LAND"}' -q 2 --tls-version
tlsv1.2-d
```

Go to the device and window in which you executed the Python script. You will see the following output:

```
Connect result: Connection Accepted.
Subscribed with QoS: 2
Topic: test/drone01. Payload: {"COMMAND": "TAKE_OFF"}
Topic: test/drone01. Payload: {"COMMAND": "LAND"}
```

The Python program successfully established a secured and encrypted connection with the MQTT server and became a subscriber to the `test/drone01` topic with a granted QoS level of 2. The program displayed the two messages it has received in the `test/drone01` topic.

Press *Ctrl+C* to stop the execution of the program. The generated MQTT client will close the connection with the MQTT server.

Creating a class to represent a drone

Now, we will write Python code that will run on the different IoT boards. Remember to copy the files we have created in the previous chapter to a directory in the computer or device that will represent the board that controls a drone and that we will use to run a Python script. We saved the files in a directory called certificates. Create a `python_certificates` directory in the computer or board you are going to use as the MQTT client for this example. Copy the following three files to this new directory:

- `ca.crt`: Certificate authority certificate file
- `device001.crt`: Client certificate file
- `device001.key`: Client key

We will create the following two classes:

- `Drone`: This class will represent a drone and provides methods that will be called whenever a command has to be processed. In order to keep the example simple, our methods will just print the actions that the drone executes after each method is called to the console output. A real-life class that represents a drone would interact with the actuators, sensors, and the other different components of the drone, whenever each method is called.
- `DroneCommandProcessor`: This class will represent a command processor that will establish a connection with an MQTT server, subscribe to a topic in which the MQTT client will receive messages with commands, analyze the incoming messages, and delegate the execution of the commands to an associated instance of the `Drone` class. The `DroneCommandProcessor` class will declare many static methods that we will specify as the callbacks for the MQTT client.

The following lines declare many variables with the values that identify each of the supported commands for the drone. In addition, the code declares many variables with the key string we will use to specify a command and the key we will use to specify a successfully processed command. All these variables are defined with all letters uppercase because we will use them as constants. Save the code in a file named `commands.py`. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_03` folder, in the `command.py` file:

```
# Key strings
COMMAND_KEY = "COMMAND"
SUCCESSFULLY_PROCESSED_COMMAND_KEY = "SUCCESSFULLY_PROCESSED_COMMAND"
# Command strings
CMD_TAKE_OFF = "TAKE_OFF"
CMD_LAND = "LAND"
CMD_LAND_IN_SAFE_PLACE = "LAND_IN_SAFE_PLACE"
CMD_MOVE_UP = "MOVE_UP"
CMD_MOVE_DOWN = "MOVE_DOWN"
CMD_MOVE_FORWARD = "MOVE_FORWARD"
CMD_MOVE_BACK = "MOVE_BACK"
CMD_MOVE_LEFT = "MOVE_LEFT"
CMD_MOVE_RIGHT = "MOVE_RIGHT"
CMD_ROTATE_RIGHT = "ROTATE_RIGHT"
CMD_ROTATE_LEFT = "ROTATE_LEFT"
CMD_SET_MAX_ALTITUDE = "SET_MAX_ALTITUDE"
CMD_SET_MIN_ALTITUDE = "SET_MIN_ALTITUDE"
```

```
KEY_DEGREES = "DEGREES"
KEY_FEET = "FEET"
```

The following lines declare the necessary imports and the same variables we used in the previous example to establish the connection with the MQTT server. You have to replace the value /Users/gaston/certificates in the certificates_path string with the path to the python_certificates directory you created. In addition, replace the value for the mqtt_server_host with the IP address for the Mosquitto server or other MQTT server you might decide to use. The code file for the sample is included in the mqtt_essentials_gaston_hillar_03 folder, in the mqtt_essentials_example03_02.py file:

```
from command import *
import paho.mqtt.client as mqtt
import os.path
import time
import json

# Replace /Users/gaston/certificates with the path
# in which you saved the certificate authority file,
# the client certificate file and the client key
certificates_path = "/Users/gaston/certificates"
ca_certificate = os.path.join(certificates_path, "ca.crt")
client_certificate = os.path.join(certificates_path, "device001.crt")
# Replace localhost with the IP for the Mosquitto
# or other MQTT server
mqtt_server_host = "localhost"
mqtt_server_port = 8883
mqtt_keepalive = 60
```

The COMMAND_KEY variable defines the key string that defines what the code will understand as the command. Whenever we receive a message that includes the specified key string, we know that the value associated with this key in the dictionary will indicate the command that the message wants the code running in the board to be processed. The MQTT client won't receive messages as dictionaries, and therefore, it is necessary to convert them from strings to dictionaries when they are not just a string.

The SUCCESSFULLY_PROCESSED_COMMAND_KEY variable defines the key string that defines what the code will use as a successfully processed command key in a response message published to the appropriate topic. Whenever we publish a message that includes the specified key string, we know that the value associated to this key in the dictionary will indicate the command that the board has successfully processed.

The following lines declare the Drone class. The code file for the sample is included in the mqtt_essentials_gaston_hillar_03 folder, in the mqtt_essentials_example03_02.py file:

```
class Drone:
    def __init__(self, name):
        self.name = name
        self.min_altitude = 0
        self.max_altitude = 30
```

```

def print_with_name_prefix(self, message):
    print("{}: {}".format(self.name, message))

def take_off(self):
    self.print_with_name_prefix("Taking off")

def land(self):
    self.print_with_name_prefix("Landing")

def land_in_safe_place(self):
    self.print_with_name_prefix("Landing in a safe place")

def move_up(self):
    self.print_with_name_prefix("Moving up")

def move_down(self):
    self.print_with_name_prefix("Moving down")

def move_forward(self):
    self.print_with_name_prefix("Moving forward")

def move_back(self):
    self.print_with_name_prefix("Moving back")

def move_left(self):
    self.print_with_name_prefix("Moving left")

def move_right(self):
    self.print_with_name_prefix("Moving right")

def rotate_right(self, degrees):
    self.print_with_name_prefix("Rotating right {} degrees".format(degrees))

def rotate_left(self, degrees):
    self.print_with_name_prefix("Rotating left {} degrees".format(degrees))

def set_max_altitude(self, feet):
    self.max_altitude = feet
    self.print_with_name_prefix("Setting maximum altitude to {} feet".format(feet))

def set_min_altitude(self, feet):
    self.min_altitude = feet
    self.print_with_name_prefix("Setting minimum altitude to {} feet".format(feet))

```

We have to specify the drone's name in the `name` required argument. The constructor, that is, the `__init__` method, saves the received `name` in an attribute with the same name. Then, the constructor sets the initial values for two attributes: `min_altitude` and `max_altitude`.

The `Drone` class declares the `print_with_name_prefix` method, which receives a message in the `message` argument and prints it with the value saved in the `name` attribute as a prefix. The other methods defined in this class call the `print_with_name_prefix` method to print messages indicating the actions

that the drone is doing with the drone's name as a prefix.

Receiving messages with Python

We will use the recently installed paho-mqtt module to subscribe to a specific topic and run code when we receive messages in the topic. We will create a DroneCommandProcessor class to represent a command processor associated to an instance of the previously coded Drone class, configure the MQTT client, the subscription to the client, and declare the code for the callbacks that are going to be executed when certain events are fired.

We will split the code for the DroneCommandProcessor class in many code snippets to make it easier to understand each code section. The following lines declare the DroneCommandProcessor class and its constructor, that is, the `__init__` method. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_03` folder, in the `mqtt_essentials_example03_02.py` file:

```
class DroneCommandProcessor:  
    commands_topic = ""  
    processed_commands_topic = ""  
    active_instance = None  
  
    def __init__(self, name, drone):  
        self.name = name  
        self.drone = drone  
        DroneCommandProcessor.commands_topic = "commands/{}".format(self.name)  
        DroneCommandProcessor.processed_commands_topic =  
"processedcommands/{}".format(self.name)  
        self.client = mqtt.Client(protocol=mqtt.MQTTv311)  
        DroneCommandProcessor.active_instance = self  
        self.client.on_connect = DroneCommandProcessor.on_connect  
        self.client.on_message = DroneCommandProcessor.on_message  
        self.client.tls_set(ca_certs = ca_certificate,  
                           certfile=client_certificate,  
                           keyfile=client_key)  
        self.client.connect(host=mqtt_server_host,  
                           port=mqtt_server_port,  
                           keepalive=mqtt_keepalive)
```

We have to specify a name for the command processor and the `drone` instance that the command processor will control in the `name` and `drone` required arguments. The constructor, that is, the `__init__` method, saves the received `name` and `drone` in attributes with the same names. Then, the constructor sets the values for the `commands_topic` and `processed_commands_topic` class attributes. The constructor uses the received `name` to determine the topic name for the commands and for the successfully processed commands, based on the specifications we discussed earlier. The MQTT client will receive messages in the topic name saved in the `command_topic` class attribute and will publish messages to the topic name saved in the `processed_commands_topic` class attribute.

Then, the constructor creates an instance of the `mqtt.Client` class (`paho.mqtt.client.Client`) that represents an MQTT client and that we will use to communicate with an MQTT broker and save it to the `client` attribute (`self.client`). As it happened in our previous example, we want to work with MQTT version 3.11, and therefore, we specified `mqtt.MQTTv311` as the value for the `protocol` argument.

The code also saves a reference to this instance in the `active_instance` class attribute because we have to access the instance in the static methods that the constructor will specify as callbacks for the different events that the MQTT client fires. We want to have all the methods related to the drone command processor in the `DroneCommandProcessor` class.

Then, the code assigns the `on_connect` static method (`DroneCommandProcessor.on_connect`) to the `self.client.on_connect` attribute and the `on_message` static method (`DroneCommandProcessor.on_message`) to the `self.client.on_message` attribute. Static methods do not receive either `self` or `cls` as the first argument, and therefore, we can use them as callbacks with the required number of arguments.

The call to the `self.client.tls_set` method configures encryption and authentication options. Finally, the constructor calls the `client.connect` method and specifies the values for the `host`, `port`, and `keepalive` arguments. This way, the code asks the MQTT client to establish a connection to the specified MQTT server. Remember that the `connect` method runs with a synchronous execution, and therefore, it is a blocking call.

The following lines declare the `on_connect` static method that is part of the `DroneCommandProcessor` class. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_03` folder, in the `mqtt_essentials_example03_02.py` file:

```
@staticmethod
def on_connect(client, userdata, flags, rc):
    print("Connected to the MQTT server")
    client.subscribe(
        DroneCommandProcessor.commands_topic,
        qos=2)
    client.publish(
        topic=DroneCommandProcessor.processed_commands_topic,
        payload="{} is listening to
messages".format(DroneCommandProcessor.active_instance.name))
```

After a connection has been successfully established with the MQTT server, the specified callback in the `self.client.on_connect` attribute will be executed, that is, the `on_connect` static method (marked with the `@staticmethod` decorator). This static method receives the `mqtt.Client` instance, which established the connection with the MQTT server in the `client` argument. The code calls the `client.subscribe` method with `DroneCommandProcessor.commands_topic` as an argument to subscribe to the topic specified in the `commands_topic` class attribute and specifies a QoS level of 2 for the subscription.

Finally, the code in the `on_connect` static method calls the `client.publish` method with `DroneCommandProcessor.processed_commands_topic` as the `topic` argument and a message string in the `payload` argument. This way, we publish a string message that specifies that the drone is listening to messages in the topic related to successfully processed commands. The message starts with the drone name.

The following lines declare the `on_message` static method that is part of the `DroneCommandProcessor`

class. In this case, the code uses a chain of if and elif statements to make it easier to understand the flow. However, a lookup table and a dictionary would provide an improved version. The code file for the sample is included in the mqtt_essentials_gaston_hillar_03 folder, in the mqtt_essentials_example03_02.py file:

```
@staticmethod
def on_message(client, userdata, msg):
    payload_string = msg.payload.decode('utf-8')
    if msg.topic == DroneCommandProcessor.commands_topic:
        print("I've received the following message:
{0}".format(str(msg.payload_string)))
    try:
        message_dictionary = json.loads(msg.payload_string)
        if COMMAND_KEY in message_dictionary:
            command = message_dictionary[COMMAND_KEY]
            drone = DroneCommandProcessor.active_instance.drone
            is_command_processed = False
            if command == CMD_TAKE_OFF:
                drone.take_off()
                is_command_processed = True
            elif command == CMD_LAND:
                drone.land()
                is_command_processed = True
            elif command == CMD_LAND_IN_SAFE_PLACE:
                drone.land_in_safe_place()
                is_command_processed = True
            elif command == CMD_MOVE_UP:
                drone.move_up()
                is_command_processed = True
            elif command == CMD_MOVE_DOWN:
                drone.move_down()
                is_command_processed = True
            elif command == CMD_MOVE_FORWARD:
                drone.move_forward()
                is_command_processed = True
            elif command == CMD_MOVE_BACK:
                drone.move_back()
                is_command_processed = True
            elif command == CMD_MOVE_LEFT:
                drone.move_left()
                is_command_processed = True
            elif command == CMD_MOVE_RIGHT:
                drone.move_right()
                is_command_processed = True
            elif command == CMD_ROTATE_RIGHT:
                degrees = message_dictionary[KEY_DEGREES]
                drone.rotate_right(degrees)
                is_command_processed = True
            elif command == CMD_ROTATE_LEFT:
                degrees = message_dictionary[KEY_DEGREES]
                drone.rotate_left(degrees)
                is_command_processed = True
            elif command == CMD_SET_MAX_ALTITUDE:
                feet = message_dictionary[KEY_FEET]
                drone.set_max_altitude(feet)
                is_command_processed = True
```

```

        elif command == CMD_SET_MIN_ALTITUDE:
            feet = message_dictionary[KEY_FEET]
            drone.set_min_altitude(feet)
            is_command_processed = True
        if is_command_processed:

DroneCommandProcessor.active_instance.publish_response_message(
    message_dictionary)
else:
    print("The message includes an unknown command.")
except ValueError:
    # msg is not a dictionary
    # No JSON object could be decoded
    print("The message doesn't include a valid command.")

```

Whenever there is a new message received in the topic saved in the `commands_topic` class attribute to which we have subscribed, the specified callback in the `self.client.on_message` attribute will be executed, that is, the `on_message` static method (marked with the `@staticmethod` decorator). This static method receives the `mqtt.Client` instance that established the connection with the MQTT broker in the `client` argument and an `mqtt.MQTTMessage` instance in the `msg` argument. The `mqtt.MQTTMessage` class describes an incoming message. First, the static method checks whether the `msg.topic` attribute, which indicates the topic in which the message has been received, matches the value in the `commands_topic` class attribute. In this case, whenever the `on_message` method is executed, the value in `msg.topic` will always match the value in the `topic` class attribute because we just subscribed to one topic. However, if we subscribe to more than one topic, it is always necessary to check in which topic the message was sent and in which one we are receiving the message. Hence, we included the code to give a clear idea of how to check the `topic` for the received message.

The code prints the string representation of the payload for the message that has been received, that is, the `msg.payload` attribute decoded to an UTF-8 string and saved in the `payload_string` variable. Then, the code assigns the result of the `json.loads` function to deserialize `payload_string` to a Python object and assigns the results to the `message_dictionary` local variable. In case the contents of `payload_string` are not JSON, a `ValueError` exception will be captured; the code will print a message indicating that the message doesn't include a valid command and no more code will be executed in the static method. In case the contents of `payload_string` are JSON, we will have a dictionary in the `message_dictionary` local variable.

Then, the code checks whether the value saved in the `COMMAND_KEY` string is included in the `message_dictionary` dictionary. If the expression evaluates to `True`, it means that the JSON message converted to a dictionary includes a command that we have to process. However, before we can process the command, we have to check which is the command, and therefore, it is necessary to retrieve the value associated with the key equivalent to the value saved in the `COMMAND_KEY` string. The code is capable of running specific code when the value is any of the commands that we received as requirements.

The code uses the `active_instance` class attribute that has a reference to the active `DroneCommandProcessor` instance to call the necessary methods for either `drone`, based on the command that has to be processed. We had to declare the callbacks as static methods, and therefore, we use this

class attribute to access the active instance. Once the command has been successfully processed, the code sets the `is_command_processed` flag to `True`. Finally, the code checks the value of this flag, and if it is equal to `True`, the code calls the `publish_response_message` for the `DroneCommandProcessor` instance saved in the `active_instance` class attribute.

Tip

Of course, in a real-life example, we should add more validations. The previous code is simplified to allow us to keep our focus on MQTT.

The following lines declare the `publish_response_message` method that is a part of the `DroneCommandProcessor` class. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_03` folder, in the `mqtt_essentials_example03_02.py` file:

```
def publish_response_message(self, message):
    response_message = json.dumps({
        SUCCESSFULLY_PROCESSED_COMMAND_KEY:
            message[COMMAND_KEY]})
    result = self.client.publish(
        topic=self.__class__.processed_commands_topic,
        payload=response_message)
    return result
```

The `publish_response_message` method receives the message dictionary that has been received with the command in the `message` argument. The method calls the `json.dumps` function to serialize a dictionary to a JSON formatted string with the response message that indicates that the command has been successfully processed. Finally, the code calls the `client.publish` method with the `processed_commands_topic` variable as the `topic` argument and the JSON formatted string (`response_message`) in the `payload` argument.

Tip

In this case, we are not evaluating the response received from the `publish` method. In addition, we are using the default value for the `qos` argument that specifies the desired quality of service. Hence, we will publish this message with a QoS level equal to 0. In more advanced scenarios, we should add code to check the results of the method and add code to the `on_publish` callback that is fired when a message is successfully published, as we did in our previous example. In this case, we use QoS level 2 for the messages that we receive with the commands.

The following lines declare the `process_commands` method, which is a part of the `DroneCommandProcessor` class. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_03` folder, in the `mqtt_essentials_example03_02.py` file:

```
def process_commands(self):
    self.client.loop()
```

The `process_commands` method calls the `loop` method for the MQTT client and ensures that

communication with the MQTT server is carried out. Think about the call to the `loop` method as synchronizing your mailbox. Any pending messages to be published in the outgoing box will be sent and any incoming messages will arrive to the inbox and the events that we have previously analyzed will be fired. This way, the drone command processor will receive messages and process commands.

Finally, the following lines declare the main block of code. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_03` folder, in the `mqtt_essentials_example03_02.py` file:

```
if __name__ == "__main__":
    drone = Drone("drone01")
    drone_command_processor = DroneCommandProcessor("drone01", drone)
    while True:
        # Process messages and the commands every 1 second
        drone_command_processor.process_commands()
        time.sleep(1)
```

The `__main__` method creates an instance of the `Drone` class named `drone` with "drone01" as the value for the `name` argument. The next line creates an instance of the `DroneCommandProcessor` class named `drone_command_processor` with "drone01" and the previously created `Drone` instance, `drone`, as the values for the `name` and `drone` arguments. This way, `drone_command_processor` will delegate the execution of the commands to the instance methods in `drone`.

The constructor for the `DroneCommandProcessor` class will subscribe to the `commands/drone01` topic in the MQTT server, and therefore, we must publish messages to this topic in order to send the commands that the code will process. In addition, the constructor will publish a message to the `processedcommands/drone01` topic. Whenever a command is successfully processed, new messages will be published to the `processedcommands/drone01` topic. Hence, we must subscribe to this topic to check the commands that the drone has executed.

The `while` loop calls the `drone_command_processor.process_commands` method and sleeps for one second. The `process_commands` method calls the `loop` method for the MQTT client to ensure that communication with the MQTT server is carried out.

Tip

There is also a threaded interface that we can run by calling the `loop_start` method for the MQTT client. This way, we can avoid multiple calls to the `loop` method. However, we call the `loop` method to make it easier to debug the code and understand how everything works under the hood.

Processing commands with Python

Now, we want to make a drone process the following commands:

```
{"COMMAND": "SET_MAX_ALTITUDE", "FEET": 20}  
{"COMMAND": "SET_MIN_ALTITUDE", "FEET": 10}  
{"COMMAND": "TAKE_OFF"}  
{"COMMAND": "MOVE_UP"}  
{"COMMAND": "ROTATE_RIGHT", "DEGREES": 90}  
{"COMMAND": "MOVE_RIGHT"}  
{"COMMAND": "LAND"}
```

Make sure the Mosquitto server or any other MQTT server you might want to use for this example is running.

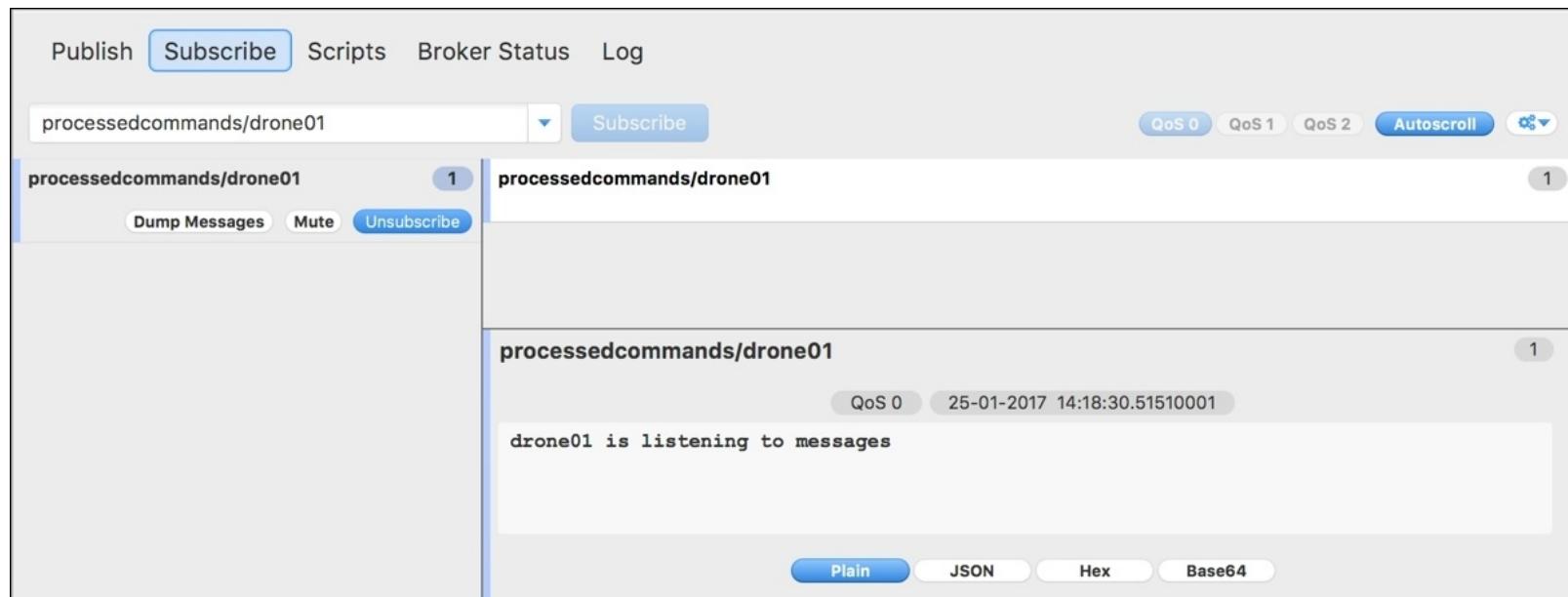
Launch MQTT.fx and follow all the steps explained in the previous chapter to configure a connection with TLS and TLS authentication. Then, click on the **Connect** button.

Click **Subscribe** and enter `processed_commands/drone01` in the dropdown on the left-hand side of the **Subscribe** button. Then, click the **Subscribe** button. MQTT.fx will display a new panel on the left-hand side with the topic filter to which we have subscribed with a QoS level of 0.

Then, execute the following line to start the example in any computer or device that you want to use as the MQTT client: `python mqtt_essentials_example03_02.py`

Keep the code running in your local computer or in the IoT board you have chosen to use for this example.

Go back to MQTT.fx and you will see that a message has arrived to the `processedcommands/drone01` topic with the following payload: `drone01 is listening to messages`. The following screenshot shows the received message in MQTT.fx:



In MQTT.fx, click **Publish** and enter commands/drone01 in the dropdown on the left-hand side of the **Publish** button. Click **QoS 2** because we want to use a QoS level of 2.

Enter the following text in the textbox below the **Publish** button: `{"COMMAND": "SET_MAX_ALTITUDE", "FEET": 20}`.

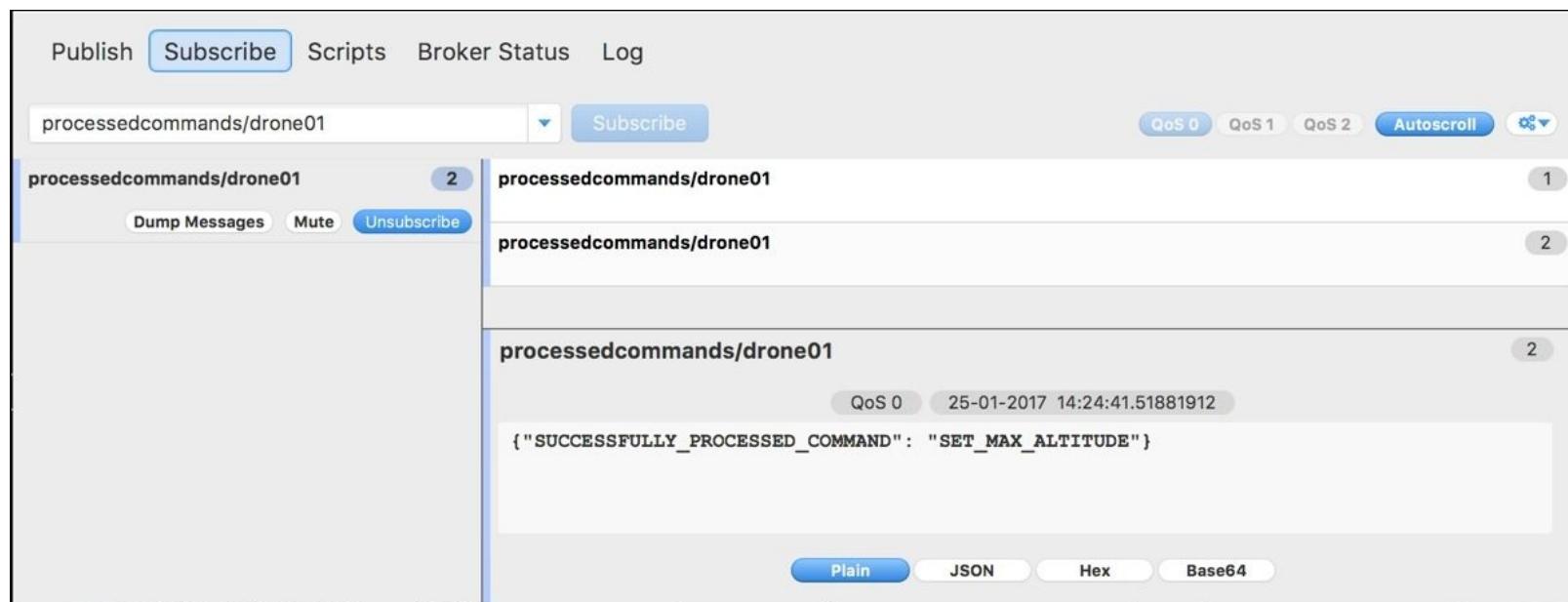
Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic with a QoS level of 2.

Go to the window in which you can see the output generated by the Python code. In case you are running the code in an IoT board, you might be using an SSH Terminal. In case you are running the code on your local computer, go to the Terminal or Command Prompt, based on the operating system you are using. You will see the following output: **Connected to the MQTT server I've received the following message: {"COMMAND": "SET_MAX_ALTITUDE", "FEET": 20} drone01: Setting maximum altitude to 20 feet**

The code has received the message with the command, the Drone instance executed the `set_max_altitude` method with 20 as an argument, and the output displays the results of executing this code.

Go back to MQTT.fx, click **Subscribe**, and you will see a new message has arrived to the `processedcommands/drone01` topic with the following payload:

`{"SUCCESSFULLY_PROCESSED_COMMAND": "SET_MAX_ALTITUDE"}`. The following screenshot shows the received message in MQTT.fx:



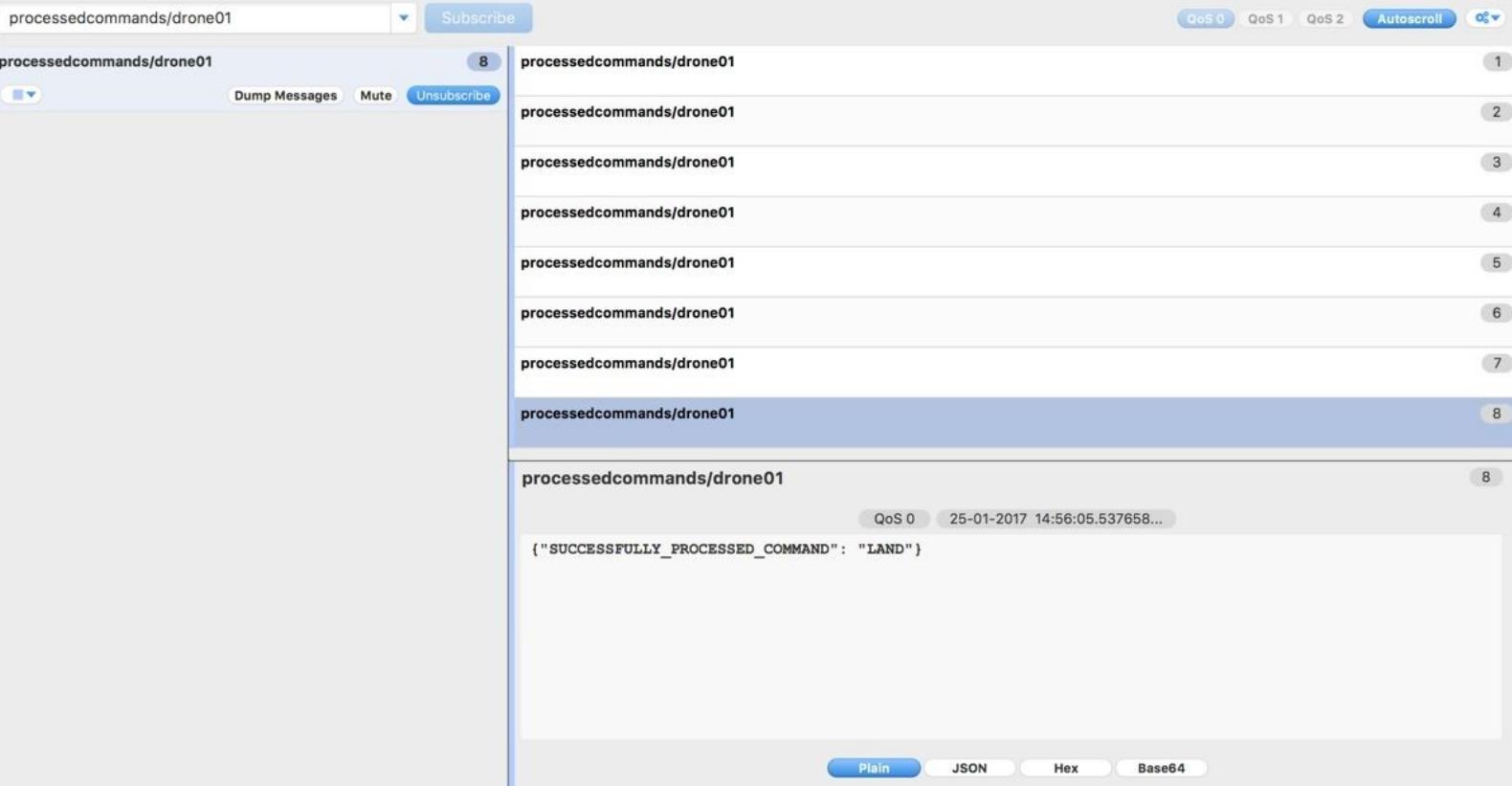
Now, repeat the following procedure for each of the commands listed below. Remove the existing text and

enter the text for the JSON string in the textbox below the **Publish** button and then click the **Publish** button. MQTT.fx will publish the entered text to the specified topic with a QoS level of 2:

- {"COMMAND": "SET_MIN_ALTITUDE", "FEET": 10}
- {"COMMAND": "TAKE_OFF"}
- {"COMMAND": "MOVE_UP"}
- {"COMMAND": "ROTATE_RIGHT", "DEGREES": 90}
- {"COMMAND": "MOVE_RIGHT"}
- {"COMMAND": "LAND"}

Go to the window in which you can see the output generated by the Python code. You will see the following output, which indicates that all the commands have been received and processed: **Connected to the MQTT server I've received the following message: {"COMMAND": "SET_MAX_ALTITUDE", "FEET": 20}** drone01: Setting maximum altitude to 20 feet I've received the following message: {"COMMAND": "SET_MIN_ALTITUDE", "FEET": 10} drone01: Setting minimum altitude to 10 feet I've received the following message: {"COMMAND": "TAKE_OFF"} drone01: Taking off I've received the following message: {"COMMAND": "MOVE_UP"} drone01: Moving up I've received the following message: {"COMMAND": "ROTATE_RIGHT", "DEGREES": 90} drone01: Rotating right 90 degrees I've received the following message: {"COMMAND": "MOVE_RIGHT"} drone01: Moving right I've received the following message: {"COMMAND": "LAND"} drone01: Landing

Go back to MQTT.fx, click **Subscribe**, and you will see a total of eight messages that have arrived to the **processedcommands/drone01** topic. You can easily check the contents of the payload for each of the received messages by clicking on the panel that represents each of them on the right-hand side of the window. The following screenshot shows the last received message in MQTT.fx:



Tip

MQTT.fx allows us to write scripts in JavaScript that make it easy to automate publishing multiple messages to topics.

Now, we will use the Mosquitto command-line utilities to subscribe to the `processedcommands/drone01` topic and publish many JSON strings with the commands to the `commands/drone01` topic. This time, we will publish the following commands:

```
{ "COMMAND": "SET_MAX_ALTITUDE", "FEET": 100 } { "COMMAND": "SET_MIN_ALTITUDE", "FEET": 20 }
{ "COMMAND": "TAKE_OFF" } { "COMMAND": "MOVE_UP" } { "COMMAND": "MOVE_RIGHT" }
{ "COMMAND": "MOVE_RIGHT" } { "COMMAND": "MOVE_LEFT" } { "COMMAND": "MOVE_LEFT" }
{ "COMMAND": "LAND" }
```

Open another Terminal in macOS or Linux, or another Command Prompt in Windows, go to the directory in which Mosquitto is installed, and run the following command. Replace `localhost` with the IP for the MQTT server. Remember to replace `ca.crt`, `device001.crt`, and `device001.key` with the full path to these files created in the `certificates` directory. Leave the command running and it will display all the messages received in the `processedcommands/drone01` topic:

```
mosquitto_sub -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t processedcommands/drone01 --tls-version tlsv1.2
```

Open another Terminal in macOS or Linux, or another Command Prompt in Windows, go to the directory in which Mosquitto is installed, and run the following commands to publish messages with commands to the commands/drone01 topic with a QoS level of 2. Make the replacements previously explained for the mosquitto_sub command:

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t commands/drone01 -q 2 --tls-version tlsv1.2 -m '{"COMMAND": "SET_MAX_ALTITUDE", "FEET": 100}'
```

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t commands/drone01 -q 2 --tls-version tlsv1.2 -m '{"COMMAND": "SET_MIN_ALTITUDE", "FEET": 20}'
```

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t commands/drone01 -q 2 --tls-version tlsv1.2 -m '{"COMMAND": "TAKE_OFF"}'
```

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t commands/drone01 -q 2 --tls-version tlsv1.2 -m '{"COMMAND": "MOVE_UP"}'
```

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t commands/drone01 -q 2 --tls-version tlsv1.2 -m '{"COMMAND": "MOVE_RIGHT"}'
```

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t commands/drone01 -q 2 --tls-version tlsv1.2 -m '{"COMMAND": "MOVE_RIGHT"}'
```

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t commands/drone01 -q 2 --tls-version tlsv1.2 -m '{"COMMAND": "MOVE_LEFT"}'
```

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t commands/drone01 -q 2 --tls-version tlsv1.2 -m '{"COMMAND": "MOVE_LEFT"}'
```

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t commands/drone01 -q 2 --tls-version tlsv1.2 -m '{"COMMAND": "LAND"}'
```

After you run the previous commands, the DroneCommandProcessor class will receive these commands and process them. After a few seconds, you will see the following output in the window that is executing the mosquitto_sub utility: **drone01 is listening to messages**

```
{"SUCCESSFULLY_PROCESSED_COMMAND": "SET_MAX_ALTITUDE"}  

{"SUCCESSFULLY_PROCESSED_COMMAND": "SET_MIN_ALTITUDE"}  

{"SUCCESSFULLY_PROCESSED_COMMAND": "TAKE_OFF"}  

{"SUCCESSFULLY_PROCESSED_COMMAND": "MOVE_UP"}  

{"SUCCESSFULLY_PROCESSED_COMMAND": "MOVE_RIGHT"}  

{"SUCCESSFULLY_PROCESSED_COMMAND": "MOVE_RIGHT"}  

{"SUCCESSFULLY_PROCESSED_COMMAND": "MOVE_LEFT"}  

{"SUCCESSFULLY_PROCESSED_COMMAND": "MOVE_LEFT"}  

{"SUCCESSFULLY_PROCESSED_COMMAND": "LAND"}
```

Go to the window in which you can see the output generated by the Python code. You will see the

following output, which indicates that all the commands have been received and processed: I've received the following message: {"COMMAND": "SET_MAX_ALTITUDE", "FEET": 100} drone01: Setting maximum altitude to 100 feet I've received the following message: {"COMMAND": "SET_MIN_ALTITUDE", "FEET": 20} drone01: Setting minimum altitude to 20 feet I've received the following message: {"COMMAND": "TAKE_OFF"} drone01: Taking off I've received the following message: {"COMMAND": "MOVE_UP"} drone01: Moving up I've received the following message: {"COMMAND": "MOVE_RIGHT"} drone01: Moving right I've received the following message: {"COMMAND": "MOVE_RIGHT"} drone01: Moving right I've received the following message: {"COMMAND": "MOVE_LEFT"} drone01: Moving left I've received the following message: {"COMMAND": "MOVE_LEFT"} drone01: Moving left I've received the following message: {"COMMAND": "LAND"} drone01: Landing

Sending messages with Python

We are coding in Python, and therefore, it makes sense to write code in Python to publish the commands to control each drone and check the results of the execution of these commands. Of course, GUI utilities such as MQTT.fx and the Mosquitto command-line utilities are extremely useful. However, after we know that things are working as we expect, we can write the necessary code to perform tests in the same programming language we are using to run the code on the IoT board.

We already wrote Python code that established a connection with the MQTT server, subscribed to a topic, and printed the received messages. Now, we are going to code a Python client that will publish messages to the `commands/drone01` topic and will subscribe to the `processedcommands/drone01` topic. We will code both a publisher and a subscriber. This way, we will be able to design applications that can talk to IoT devices with MQTT messages, with Python as the programming language for the client applications. Specifically, the applications will be able to communicate through an MQTT server with Python code in all the publisher and subscriber devices.

We can run the Python client on any other computer or IoT board that is capable of executing Python 2.7.x or Python 3.x.

We will create many functions that we will assign as the callbacks to the events in the MQTT client. In addition, we will declare variables, a helper class, and a helper function to make it easy to publish a message with a command and the required values for the command. The following lines show the code that defines the variables, the helper class, and the functions. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_03` folder, in the `mqtt_essentials_example03_03.py` file:

```
from command import *
from command import *
import paho.mqtt.client as mqtt
import os.path
import time
import json

# Replace /Users/gaston/certificates with the path
# in which you saved the certificate authority file,
# the client certificate file and the client key
certificates_path = "/Users/gaston/certificates"
ca_certificate = os.path.join(certificates_path, "ca.crt")
client_certificate = os.path.join(certificates_path, "device001.crt")
client_key = os.path.join(certificates_path, "device001.key")
mqtt_server_host = "localhost"
mqtt_server_port = 8883
mqtt_keepalive = 60

drone_name = "drone01"
commands_topic = "commands/{}".format(drone_name)
processed_commands_topic = "processedcommands/{}".format(drone_name)

class LoopControl:
```

```

is_last_command_processed = False

def on_connect(client, userdata, rc):
    print("Connect result: {}".format(mqtt.connack_string(rc)))
    client.subscribe(processed_commands_topic)

def on_message(client, userdata, msg):
    if msg.topic == processed_commands_topic:
        payload_string = msg.payload.decode('utf-8')
        print(str(msg.payload_string))
        if str(msg.payload_string).count(CMD_LAND_IN_SAFE_PLACE) > 0:
            LoopControl.is_last_command_processed = True

def on_subscribe(client, userdata, mid, granted_qos):
    print("Subscribed with QoS: {}".format(granted_qos[0]))

def publish_command(client, command_name, key="", value ""):
    if key:
        command_message = json.dumps({
            COMMAND_KEY: command_name,
            key: value})
    else:
        command_message = json.dumps({
            COMMAND_KEY: command_name})
    result = client.publish(topic=commands_topic,
                           payload=command_message, qos=2)
    return result

```

The code declares the same variables we used in the previous examples for the certificates path and files, and the configuration for the MQTT server. Then, the code declares the `drone_name` variable that saves a string with "drone01" and that we can easily replace with the name of the drone that we want to control. Our main goal is to build and publish command messages to the topic specified in the `commands_topic` variable. We will subscribe to the topic specified in the `processed_commands_topic` variable.

The `LoopControl` class declares a class attribute named `is_last_command_processed`, initialized to `False`. We will use this class attribute as a flag to control the network loop.

The `on_connect` function is the callback that will be executed once a successful connection has been established with the MQTT server. The code calls the `subscribe` method for the MQTT client received in the `client` argument to subscribe to the topic name saved in the `processed_commands_topic` with a QoS level of 0.

The `on_message` function will be executed every time a new message arrives to the topic to which we have subscribed. The function just prints the raw string with the payload of the received message. If the string representation of the payload includes the string saved in the `CMD_LAND_IN_SAFE_PLACE` constant, we assume that the last command was successfully executed and the code sets the `LoopControl.is_last_command_processed` to `True`.

The `on_subscribe` function will be called when the subscription has been successfully completed.

The `publish_command` function receives the MQTT client, the command name, the key, and the value that provides the necessary information to execute the command in the `client`, `command_name`, `key`, and `value` required arguments. The function creates a dictionary and saves the results of serializing the dictionary to a JSON formatted string in the `command_message` local variable. The `COMMAND_KEY` constant is the first key for the dictionary, and `command_name` receives as an argument the value that composes the first key-value pair. Then, the code calls the `client.publish` method to publish the `command_message` JSON formatted string to the topic name saved in the `commands_topic` variable with a QoS level of 2.

Working with the network loop with Python

Now, we will use the previously coded functions to write a `__main__` method that publishes many commands included in MQTT messages that the code controlling the drone will process. The following lines show the code for the `__main__` method, that is, the main block of code. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_03` folder, in the `mqtt_essentials_example03_03.py` file:

```
if __name__ == "__main__":
    client = mqtt.Client(protocol=mqtt.MQTTv311)
    client.on_connect = on_connect
    client.on_subscribe = on_subscribe
    client.on_message = on_message
    client.tls_set(ca_certs = ca_certificate,
                  certfile=client_certificate,
                  keyfile=client_key)
    client.connect(host= mqtt_server_host,
                  port= mqtt_server_port,
                  keepalive= mqtt_keepalive)
    publish_command(client, CMD_TAKE_OFF)
    publish_command(client, CMD_MOVE_UP)
    publish_command(client, CMD_ROTATE_LEFT, KEY_DEGREES, 90)
    publish_command(client, CMD_ROTATE_LEFT, KEY_DEGREES, 45)
    publish_command(client, CMD_ROTATE_LEFT, KEY_DEGREES, 45)
    publish_command(client, CMD_LAND_IN_SAFE_PLACE)
    while LoopControl.is_last_command_processed == False:
        client.loop()
        time.sleep(1)
    client.disconnect()
    client.loop()
```

The first lines of the code are similar to the first example. After calling the `client.connect` method, the code calls the `publish_command` command many times to build and publish the messages with the commands.

The `while` loop calls the `client.loop` method to ensure that communication with the MQTT server is carried out, and then it sleeps for one second. After the last command is processed, the `LoopControl.is_last_command_processed` class variable is set to `False` and the `while` loop ends its execution. When this happens, the code calls the `client.disconnect` method and `client.loop` to make sure that the disconnect request is processed.

In this case, we don't want the loop to run forever because we have a specific goal of processing a set of commands, and then, we close the connection with the MQTT server.

Make sure that the code that controls `drone01` is running. Then, execute the following line to start the Python client example in any computer or device that you want to use as the MQTT client:

```
python mqtt_essentials_example03_03.py
```

Go to the device and window in which you executed the previous Python script. You will see the

following output. The Python code will show all the messages received in the processed_commands_topic. In this case, the program will end its execution after the drone indicates that it has successfully processed the LAND_IN_SAFE_PLACE command:

```
Connect result: Connection Accepted.  
Subscribed with QoS: 0  
{ "SUCCESSFULLY_PROCESSED_COMMAND": "TAKE_OFF" }  
{ "SUCCESSFULLY_PROCESSED_COMMAND": "MOVE_UP" }  
{ "SUCCESSFULLY_PROCESSED_COMMAND": "ROTATE_LEFT" }  
{ "SUCCESSFULLY_PROCESSED_COMMAND": "ROTATE_LEFT" }  
{ "SUCCESSFULLY_PROCESSED_COMMAND": "ROTATE_LEFT" }  
{ "SUCCESSFULLY_PROCESSED_COMMAND": "LAND_IN_SAFE_PLACE" }
```

Go to the device and window in which you executed the Python script that controlled the drone, that is, mqtt_essentials_example_03_02.py. You will see the following output:

```
I've received the following message: { "COMMAND": "TAKE_OFF" }  
drone01: Taking off  
I've received the following message: { "COMMAND": "MOVE_UP" }  
drone01: Moving up  
I've received the following message: { "DEGREES": 90, "COMMAND": "ROTATE_LEFT" }  
drone01: Rotating left 90 degrees  
I've received the following message: { "DEGREES": 45, "COMMAND": "ROTATE_LEFT" }  
drone01: Rotating left 45 degrees  
I've received the following message: { "DEGREES": 45, "COMMAND": "ROTATE_LEFT" }  
drone01: Rotating left 45 degrees  
I've received the following message: { "COMMAND": "LAND_IN_SAFE_PLACE" }  
drone01: Landing in a safe place
```

The following screenshot shows two Terminal windows running on a computer with macOS. The Terminal on the right-hand side is displaying the debug messages for a Mosquitto server that runs on the computer. The Terminal on the left-hand side displays the results of running the code for the Python client that published the commands to the drone, that is, mqtt_essentials_example_03_03.py:

```
Terminal Shell Edit View Window Help
Chapter03 — bash — 59x44
connect result: CONNECTION ACCEPTED.
Subscribed with QoS: 0
{"SUCCESSFULLY_PROCESSED_COMMAND": "TAKE_OFF"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "MOVE_UP"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "ROTATE_LEFT"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "ROTATE_LEFT"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "ROTATE_LEFT"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "LAND_IN_SAFE_PLACE"}
Gastons-MacBook-Pro:Chapter03 gaston$ * mosquitto_certs — mosquitto -c /usr/local/etc/mosquitto/mosquitto.conf -v — 69x41
1485398446: Received PUBREL from paho/4F07D57DF9FDC39319 (Mid: 4)
1485398446: Sending PUBREL to paho/4F07D57DF9FDC39319 (Mid: 4)
1485398447: Received PUBREC from paho/4F07D57DF9FDC39319 (Mid: 5)
1485398447: Sending PUBREL to paho/4F07D57DF9FDC39319 (Mid: 5)
1485398447: Received PUBLISH from paho/4F07D57DF9FDC39319 (d0, q0, r0 , m0, 'processedcommands/drone01', ... (46 bytes))
1485398447: Sending PUBLISH to paho/18C27D0DCCF6DCD3DA (d0, q0, r0, m0, 'processedcommands/drone01', ... (46 bytes))
1485398447: Received PUBCOMP from paho/4F07D57DF9FDC39319 (Mid: 1)
1485398447: Received PUBREC from paho/4F07D57DF9FDC39319 (Mid: 6)
1485398447: Sending PUBREL to paho/4F07D57DF9FDC39319 (Mid: 6)
1485398447: Received PUBLISH from paho/4F07D57DF9FDC39319 (d0, q0, r0 , m0, 'processedcommands/drone01', ... (45 bytes))
1485398447: Sending PUBLISH to paho/18C27D0DCCF6DCD3DA (d0, q0, r0, m0, 'processedcommands/drone01', ... (45 bytes))
1485398447: Received PUBCOMP from paho/4F07D57DF9FDC39319 (Mid: 2)
1485398448: Received PUBLISH from paho/4F07D57DF9FDC39319 (d0, q0, r0 , m0, 'processedcommands/drone01', ... (49 bytes))
1485398448: Sending PUBLISH to paho/18C27D0DCCF6DCD3DA (d0, q0, r0, m0, 'processedcommands/drone01', ... (49 bytes))
1485398448: Received PUBCOMP from paho/4F07D57DF9FDC39319 (Mid: 3)
1485398448: Received PUBLISH from paho/4F07D57DF9FDC39319 (d0, q0, r0 , m0, 'processedcommands/drone01', ... (49 bytes))
1485398448: Sending PUBLISH to paho/18C27D0DCCF6DCD3DA (d0, q0, r0, m0, 'processedcommands/drone01', ... (49 bytes))
1485398448: Received PUBCOMP from paho/4F07D57DF9FDC39319 (Mid: 4)
1485398448: Received PUBLISH from paho/4F07D57DF9FDC39319 (d0, q0, r0 , m0, 'processedcommands/drone01', ... (49 bytes))
1485398448: Sending PUBLISH to paho/18C27D0DCCF6DCD3DA (d0, q0, r0, m0, 'processedcommands/drone01', ... (49 bytes))
1485398448: Received PUBCOMP from paho/4F07D57DF9FDC39319 (Mid: 5)
1485398448: Received PUBLISH from paho/4F07D57DF9FDC39319 (d0, q0, r0 , m0, 'processedcommands/drone01', ... (56 bytes))
1485398448: Sending PUBLISH to paho/18C27D0DCCF6DCD3DA (d0, q0, r0, m0, 'processedcommands/drone01', ... (56 bytes))
1485398448: Received PUBCOMP from paho/4F07D57DF9FDC39319 (Mid: 6)
1485398454: Received DISCONNECT from paho/18C27D0DCCF6DCD3DA
1485398454: Client paho/18C27D0DCCF6DCD3DA disconnected.
```

The next screenshot shows a Terminal window running on the Raspbian operating system on a Raspberry Pi 3 board. The Raspberry Pi 3 is running the Python code that controls the drone, that is, `mqtt_essentials_example_03_02.py`:



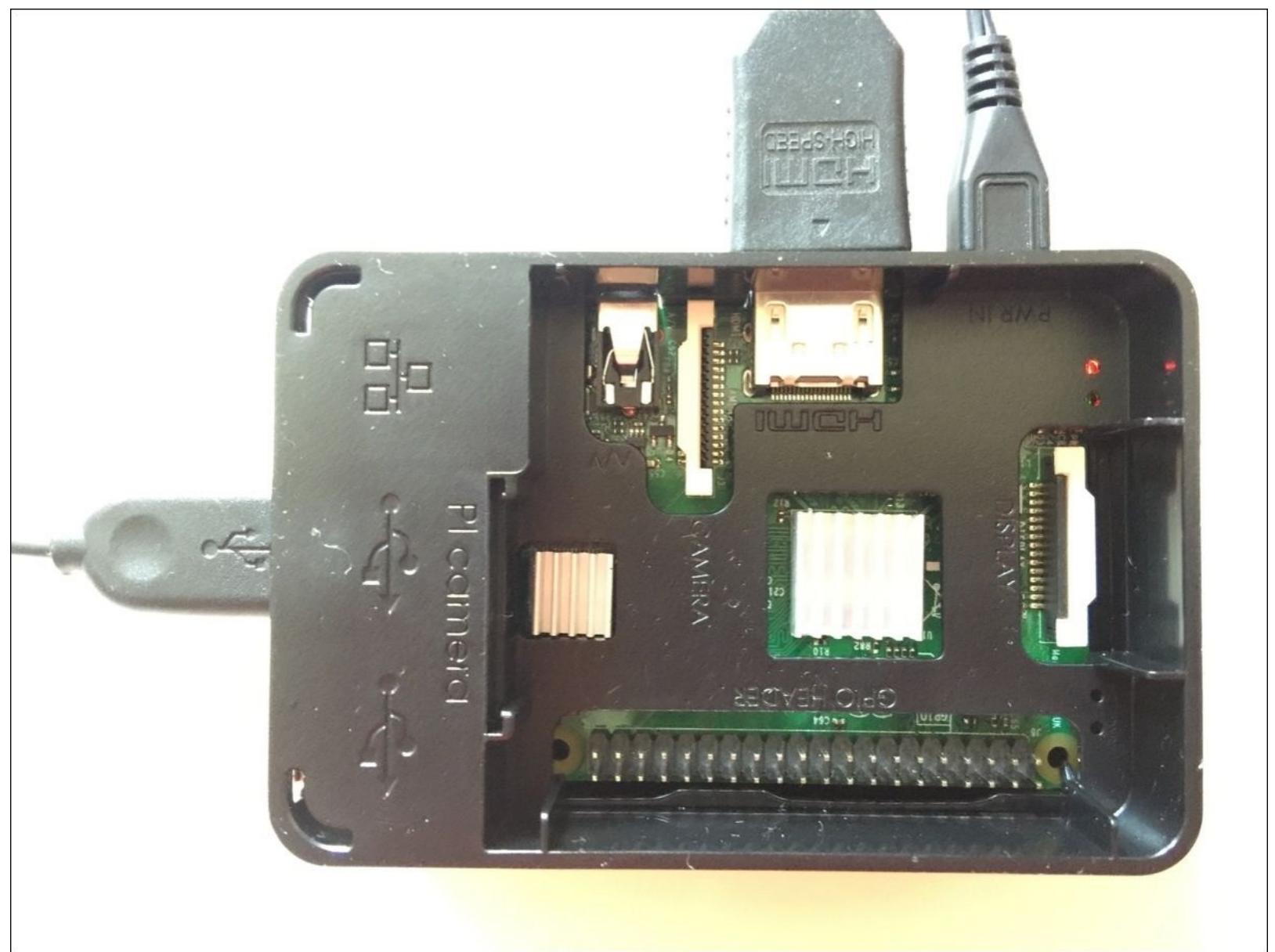
pi@raspberrypi: ~/drone

File Edit Tabs Help

```
pi@raspberrypi:~/drone $ python gaston_hillar_mqtt_03_02.py
Connected to the MQTT server
I've received the following message: {"COMMAND": "TAKE_OFF"}
drone01: Taking off
I've received the following message: {"COMMAND": "MOVE_UP"}
drone01: Moving up
I've received the following message: {"DEGREES": 90, "COMMAND": "ROTATE_LEFT"}
drone01: Rotating left 90 degrees
I've received the following message: {"DEGREES": 45, "COMMAND": "ROTATE_LEFT"}
drone01: Rotating left 45 degrees
I've received the following message: {"DEGREES": 45, "COMMAND": "ROTATE_LEFT"}
drone01: Rotating left 45 degrees
I've received the following message: {"COMMAND": "LAND_IN_SAFE_PLACE"}
drone01: Landing in a safe place
```



The next picture shows the Raspberry Pi 3 board that is running the code whose output was shown in the previous screenshot. A computer is running the Mosquitto server:



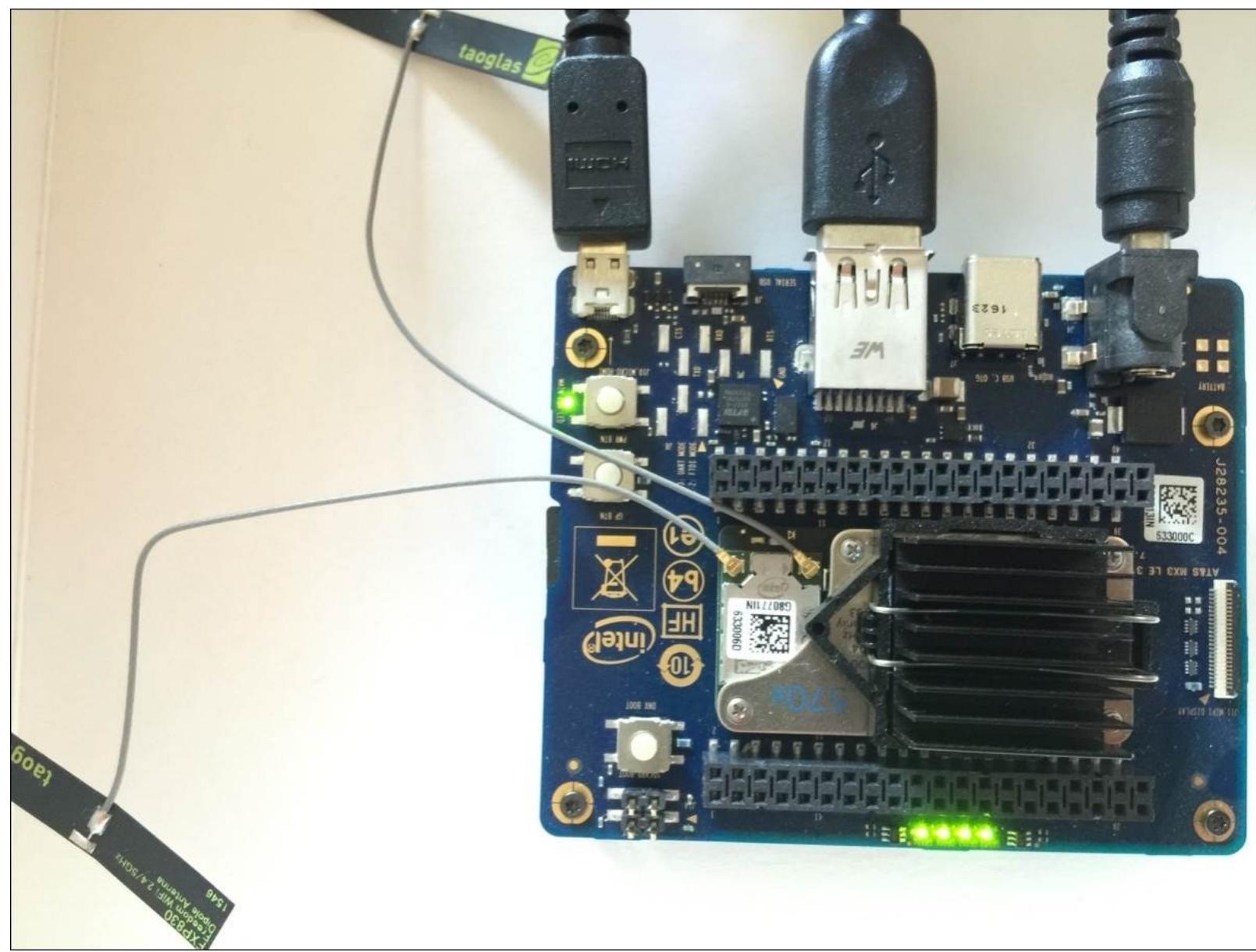
The following screenshot shows three Terminal windows running on a computer with macOS. The Terminal in the upper-left corner displays an SSH terminal that is running the Python code controlling the drone on an Intel Joule 570x board. That is, the board is running `mqtt_essentials_example_03_02.py`. The Terminal in the lower-left corner displays the results of running the code for the Python client that published the commands to the drone, that is, `mqtt_essentials_example_03_03.py`. The Terminal on the right-hand side displays the debug messages for a Mosquitto server that is running on the computer:

```

gaston — ssh root@192.168.1.108 — 79x21
root@intel-corei7-64:~/drone# echo 'Running code on Intel Joule board'
Running code on Intel Joule board
root@intel-corei7-64:~/drone# python gaston_hillar_mqtt_03_02.py
Connected to the MQTT server
I've received the following message: {"COMMAND": "TAKE_OFF"}
drone01: Taking off
I've received the following message: {"COMMAND": "MOVE_UP"}
drone01: Moving up
I've received the following message: {"DEGREES": 90, "COMMAND": "ROTATE_LEFT"}
drone01: Rotating left 90 degrees
I've received the following message: {"DEGREES": 45, "COMMAND": "ROTATE_LEFT"}
drone01: Rotating left 45 degrees
I've received the following message: {"DEGREES": 45, "COMMAND": "ROTATE_LEFT"}
drone01: Rotating left 45 degrees
I've received the following message: {"COMMAND": "LAND_IN_SAFE_PLACE"}
drone01: Landing in a safe place
d: 6)
485447731: Received PUBLISH from paho/7817AFDF6D4F497A7
(d0, q0, r0, m0, 'processedcommands/drone01', ... (45
bytes))
485447731: Sending PUBLISH to paho/77092268D99945D328 (
0, q0, r0, m0, 'processedcommands/drone01', ... (45 by
ts))
485447731: Received PUBCOMP from paho/7817AFDF6D4F497A7
(Mid: 2)
485447732: Received PUBLISH from paho/7817AFDF6D4F497A7
(d0, q0, r0, m0, 'processedcommands/drone01', ... (49
bytes))
485447732: Sending PUBLISH to paho/77092268D99945D328 (
0, q0, r0, m0, 'processedcommands/drone01', ... (49 by
ts))
485447732: Received PUBCOMP from paho/7817AFDF6D4F497A7
(Mid: 3)
485447732: Received PUBLISH from paho/7817AFDF6D4F497A7
(d0, q0, r0, m0, 'processedcommands/drone01', ... (49
bytes))
485447732: Sending PUBLISH to paho/77092268D99945D328 (
0, q0, r0, m0, 'processedcommands/drone01', ... (49 by
ts))
485447732: Received PUBCOMP from paho/7817AFDF6D4F497A7
(Mid: 4)
485447732: Received PUBLISH from paho/7817AFDF6D4F497A7
(d0, q0, r0, m0, 'processedcommands/drone01', ... (49
bytes))
485447732: Sending PUBLISH to paho/77092268D99945D328 (
0, q0, r0, m0, 'processedcommands/drone01', ... (49 by
ts))
485447732: Received PUBCOMP from paho/7817AFDF6D4F497A7
(Mid: 5)
485447732: Received PUBLISH from paho/7817AFDF6D4F497A7
(d0, q0, r0, m0, 'processedcommands/drone01', ... (56
bytes))
485447732: Sending PUBLISH to paho/77092268D99945D328 (
0, q0, r0, m0, 'processedcommands/drone01', ... (56 by
ts))
485447732: Received PUBCOMP from paho/7817AFDF6D4F497A7
(Mid: 6)
485447737: Received DISCONNECT from paho/77092268D99945
328
485447737: Client paho/77092268D99945D328 disconnected.

Gastons-MacBook-Pro:chapter03 gaston$ python gaston_hillar_mqtt_03_03.py
Connect result: Connection Accepted.
Subscribed with QoS: 0
{"SUCCESSFULLY_PROCESSED_COMMAND": "TAKE_OFF"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "MOVE_UP"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "ROTATE_LEFT"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "ROTATE_LEFT"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "ROTATE_LEFT"}
{"SUCCESSFULLY_PROCESSED_COMMAND": "LAND_IN_SAFE_PLACE"}
Gastons-MacBook-Pro:chapter03 gaston$ 
```

The next picture shows the Intel Joule 570x board that runs the code whose output was shown in the previous screenshot. Remember that the Mosquitto server is running on a computer:



Test your knowledge

1. Which of the following Python modules is the Paho Python Client?
 1. paho-mqtt.
 2. paho-client-pip.
 3. paho-python-client.
2. To establish a connection with an MQTT server that uses TLS, which method do you have to call for the `paho.mqtt.client.Client` instance before calling `connect`?
 1. `connect_with_tls`.
 2. `tls_set`.
 3. `configure_tls`.
3. After the `paho.mqtt.client.Client` instance establishes a connection with the MQTT server, the callback assigned to which of the following attributes will be called?
 1. `on_connection`.
 2. `on_connect`.
 3. `connect_callback`.
4. After the `paho.mqtt.client.Client` instance receives a message from one of the topic filters to which it has subscribed, the callback assigned to which of the following attributes will be called?
 1. `on_message_arrived`.
 2. `on_message`.
 3. `message_arrived_callback`.
5. Which of the following methods of a `paho.mqtt.client.Client` instance calls the `loop` method for us in an infinite blocking loop?
 1. `infinite_loop`.
 2. `loop_while_true`.
 3. `loop_forever`.

Summary

In this chapter, we analyzed the requirements for controlling a drone with MQTT messages. We defined the topics that we will use and the commands that will be part of the messages' payloads to control a drone. Then, we worked with the Paho Python Client to write Python code that connected an MQTT client to the MQTT server.

We understood the methods we needed to call for the Paho Python Client and their parameters. We analyzed how callback worked and we were able to publish messages to topics, subscribe to topic filters, and receive and process messages.

We wrote code that processed commands for a drone with Python. The code is able to run on different IoT platforms, including Intel Joule boards and Raspberry Pi 3, and also on any computer that is capable of executing Python 2.7.x or Python 3.x. We worked with the network loop for the MQTT client in Python. We worked with a GUI utility, command-line based utilities, and, finally, with Python code to control the drone and display the feedback provided by the drone.

Now that we understand how to use Python to work with MQTT, we will use this protocol to interact with sensors by using the Java programming language, which is what we are going to discuss in the next chapter.

Chapter 4. Interacting with Sensors in Java

In this chapter, we will write Java code to control sensors with MQTT messages. We will write code that will be able to run on different popular IoT platforms. We will:

- Understand the requirements to control sensors with MQTT
- Define the topics and commands to control sensors
- Install the Eclipse Paho Java Client
- Create a security helper class to establish a TLS secured connection
- Configure a TLS secured connection in Java
- Work with asynchronous notifications and callbacks in Java
- Subscribe to topics and understand notification events in Java
- Process commands that interact with sensors in Java
- Work with last will and testament in Java
- Publish messages with commands to target sensors

Understanding the requirements

In the previous chapters, we learned how MQTT works in detail. We understood how to establish a connection between an MQTT client and an MQTT server. We learned what happened when we subscribed to topic filters and when a publisher sent messages to specific topics. We installed and secured a Mosquitto server. Then, we controlled a drone with Python.

Now, we will use Java as our main programming language to generate MQTT clients that will act as publishers and subscribers. We will connect a Java MQTT client to the MQTT server and we will process simple commands to control sensors with MQTT messages. We will use TLS encryption and TLS authentication because we don't want any MQTT client to be able to send commands to sensors wired to IoT boards. We want our Java code to run on many platforms because we will use the same code base to control sensors that use the following IoT boards:

- Raspberry Pi 3
- Raspberry Pi 2 Model B
- Intel Edison
- Intel Galileo Gen 2
- Intel Joule 570x
- Intel Joule 550x

Each platform will work with different kinds of sensors. However, we will focus on the basic features to keep our example simple and to stay concentrated on MQTT. Then, we will be able to use this project as a baseline for other solutions that require us to run code on an IoT board that runs Java code, has to connect to an MQTT server, and has to process commands.

The code that runs on the boards that have sensors wired to them must be able to process commands received in messages in a specific topic. We will use strings with short commands in plain English in the payloads.

A client application also written in Java must be able to control one or more sensors. We will also write the client application in Java and it will publish MQTT messages with strings to the topic for each drone. The client application must display the results of executing each command. Each IoT board must publish a message to a specific topic whenever a command was successfully executed.

Defining the topics and commands

We will use the following topic name to publish the commands for the sensors wired to an IoT board: `commands/boards/boardname/sensors/sensornname` , where **boardname** must be replaced with a unique name assigned to a board and **sensornname** must be replaced with a unique name assigned to a sensor. For example, if we assign `location001` as the name for an Intel Joule 550x board and we assign `earthhumidity` as the name for a humidity sensor wired to this board, we will have to publish commands to the `commands/boards/location001/sensors/earthhumidity` topic and the code that runs on this board will subscribe to this topic to receive messages with commands and react to them. In fact, the board can subscribe to `commands/boards/location001/sensors/+` to receive messages with commands for all the sensors wired to the board named `location001`.

We will use the following topic name to make the boards publish data retrieved from the sensors: `data/boards/boardname/sensors/sensornname` , where **boardname** must be replaced with a unique name assigned to a board and **sensornname** must be replaced with a unique name assigned to a sensor. For example, if we assign `location300` as the name for a Raspberry Pi 3 board, and we assign `sunlight` as the name for a sunlight sensor wired to this board, the client that wants to receive data retrieved from this sensor has to subscribe to the `data/boards/location300/sensors/sunlight/+` topic because the sensors will provide data in different subtopics. Another option for the client is to subscribe to `data/boards/location300/sensors/#` to subscribe to all the sensors and their different subtopics. For example, the sunlight sensor will provide the following data:

- **Visible light:** The visible light will be published to the `data/boards/location300/sensors/sunlight/visiblelight` topic
- **Infrared light:** The infrared light will be published to the `data/boards/location300/sensors/sunlight/ir` topic
- **Ultraviolet index:** The ultraviolet light will be published to the `data/boards/location300/sensors/sunlight/uv` topic

The commands will be sent as strings in plain English. We will have just two valid commands:

- **TURN ON:** Turn on the sensor and make the board start transmitting the data retrieved from this sensor every 1 second to the appropriate topic
- **TURN OFF:** Turn off the sensor and make the board stop transmitting the data retrieved from this sensor

The following line shows an example of the payload for the command that turns on a sensor: **TURN ON**

The following line shows an example of the payload for the visible light value retrieved from a sunlight sensor: **250 lm**

We will use the following topic name to make the boards publish their status:

`status/boards/boardname`, where **boardname** must be replaced with a unique name assigned to a board. For example, if we assign `location005` as the name for an Intel Edison board, we will have to subscribe to the `status/boards/location005` topic and the code that runs on this board will publish its

status to this topic. When the board subscribes to the appropriate topics to listen to messages with commands, it will publish a message with the payload, `ONLINE`, to this topic. This message will be published as a retained message with a QoS level of 2. We will take advantage of the last will and testament feature to publish a message with the payload, `OFFLINE`, to this topic whenever the connection between the MQTT client running on the board and the MQTT server is lost or the application stops its execution. We analyzed the last will and testament feature in [Chapter 1](#), *Lightweight Messaging with MQTT 3.1.1 and Mosquitto*.

We have all the necessary details to start coding with Java. First, we will develop a simple Java application that subscribes to a topic and displays the received messages. Then, we will create the Java application that can control sensors based on the previously explained requirements.

Installing the Eclipse Paho Java Client

The Eclipse Paho project provides an open source client implementation of MQTT. The project includes a Java client, also known as the Paho Java Client or Eclipse Paho Java client.

The following is the web page for the Eclipse Paho project: <http://www.eclipse.org/paho>

The following is the web page for the Eclipse Paho Java client: <https://eclipse.org/paho/clients/java>

Tip

We can use the Eclipse Paho Java client in many modern IoT boards that support Java. For example, we can install and use this client on Raspberry Pi boards, such as Raspberry Pi 3, and on Intel IoT boards, such as the Intel Joule, Intel Edison, and Intel Galileo. We just need to make sure that Java is installed. You can use your development computer to run the examples or any of the mentioned boards. You will need Java 8 or a later version installed on your development computer.

This Java client provides both a blocking (synchronous) and a non-blocking (asynchronous) API. The blocking API is a wrapper of the non-blocking API that makes all the necessary things to make the code wait until each method has a result. We will work with the non-blocking (asynchronous) API because it is the one that we will use most of the times. This API is more complex than the blocking API due to the asynchronous execution of its methods and the need to work with callbacks. However, once you learn the non-blocking API, you can easily switch to the blocking API in case you need to use it.

The most complex task with this API is to establish a secure connection with an MQTT server. The Paho Java Client uses standard Java TLS/SSL mechanisms, and therefore, we will use the Bouncy Castle libraries. You can read more information about these libraries at <http://www.bouncycastle.org>.

To use the Paho Java Client in combination with the Bouncy Castle libraries, we can start a Maven project in our favorite IDE and add the following lines to the `pom.xml` file. We must add the lines before `</project>`. In case you don't want to work with Maven, you can check the documentation for the Paho Java Client and the Bouncy Castle libraries to download the appropriate versions and use them in your projects. The following XML shows the new lines that you must add to the `pom.xml` file. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01` folder:

```
<repositories>
    <repository>
        <id>Eclipse Paho Repo</id>
        <url>https://repo.eclipse.org/content/repositories/paho-
releases/</url>
    </repository>
</repositories>
<dependencies>
    <dependency>
        <groupId>org.eclipse.paho</groupId>
        <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
        <version>1.1.0</version>
    </dependency>
    <dependency>
```

```
</dependency>
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcpkix-jdk15on</artifactId>
    <version>1.56</version>
</dependency>
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcprov-jdk15on</artifactId>
    <version>1.56</version>
</dependency>
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.9</version>
</dependency>
</dependencies>
```

Note

The sample has been tested with the version numbers specified in the previous lines that are included in the pom.xml file. The code works with Java 8 and Java 9.

After you make the previous changes in the pom.xml file, the following .jar files will be added to the Maven dependencies for the Maven project:

- org.eclipse.paho.client.mqttv3-1.1.0.jar
- bcmail-jdk15on-1.56.jar
- bcpkix-jdk15on-1.56.jar
- bcprov-jdk15on-1.56.jar
- commons-codec-1.9.jar

Creating a security helper class to establish a TLS secured connection

Now, we will create a new security helper class named `SecurityHelper`, which will provide many static methods that we will use to make it easy to establish a TLS secured connection with an MQTT server. We will use the recently added dependencies for the Bouncy Castle libraries to load the certificates and key files.

The public static `CreateSocketFactory` method will receive the file names for the certificate authority certificate, the client certificate, and the client key. The method will load all these files, generate the appropriate instances from them, and return an instance of `java.net.ssl.SSLSocketFactory`.

Tip

It is very important to avoid being confused by the old SSL name. We will be working with TLS version 1.2, but Java has the old `java.net.ssl.SSLSocketFactory` that has been upgraded to work with the latest TLS versions (but it didn't change its name from SSL to TLS to keep compatibility with the previous versions).

The `SecurityHelper` class will provide the following private static methods that will make it easier to understand the code in the public static `createSocketFactory` method:

- `getKeyFactoryInstance`: This returns a `java.key.KeyFactory` instance that converts public/private keys of the RSA algorithm.
- `createX509CertificateFromFile`: This method receives a certificate file name, loads its contents, generates an instance of `java.security.cert.X509Certificate` from this file, and returns the created instance. Other methods will call the `createX509CertificateFromFile` method to generate an `X509Certificate` instance from a certificate authority certificate file or from a client certificate file.
- `createPrivateKeyFromPemFile`: This method receives a key file name in the PEM format, loads its contents, generates an instance of `java.Security.PrivateKey` from this file, and returns the created instance. This method will allow us to create a `PrivateKey` instance from the client key file.
- `createKeyManagerFactory`: This method receives a client certificate file name, a client key file name, and a client key password, and calls the previously explained `createX509CertificateFromFile` and `createPrivateKeyFromPemFile` methods to generate the `X509Certificate` and `PrivateKey` instances. Then, the method uses these instances to specify the certificate and key for an instance of the `java.security.KeyStore` instance and uses it to create and return an instance of the `java.net.ssl.KeyManagerFactory` class.
- `createTrustManagerFactory`: This method receives a certificate authority certificate file name and calls the previously explained `createX509CertificateFromFile` method to generate an `X509Certificate` instance. Then, the method uses this instance to specify the certificate authority certificate for an instance of the `java.security.KeyStore` instance and uses it to create and return an instance of the `java.net.ssl.TrustManagerFactory` class.

We will split the code for the `SecurityHelper` class into many code snippets to make it easier to understand each code section. The following lines declare all the necessary imports. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01/src/SecurityHelper.java` file:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;

import java.security.KeyFactory;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.UnrecoverableKeyException;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;

import javax.net.ssl.KeyManager;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManager;
import javax.net.ssl.TrustManagerFactory;

import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.io.pem.PemObject;
import org.bouncycastle.util.io.pem.PemReader;
```

The following lines start the declaration of the `SecurityHelper` class. The code declares the `TLS_VERSION` string constant with "TLSv1.2" as its value. We will work with TLS version 1.2 and we will use this constant when it is necessary to specify the desired version of TLS/SSL in the code. The code also declares the following three methods: `getKeyFactoryInstance`, `createX509CertificateFromFile`, and `createPrivateKeyFromPemFile` methods. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01/src/SecurityHelper.java` file:

```
public class SecurityHelper {
    private final static String TLS_VERSION = "TLSv1.2";

    private static KeyFactory getKeyFactoryInstance() throws NoSuchAlgorithmException {
        return KeyFactory.getInstance("RSA");
    }

    private static X509Certificate createX509CertificateFromFile(
```

```

    final String certificateFileName) throws IOException, CertificateException
{
    final File file = new java.io.File(certificateFileName);
    if (!file.isFile()) {
        throw new IOException(
            String.format("The certificate file %s doesn't
exist.", certificateFileName));
    }
    final CertificateFactory certificateFactoryX509 =
CertificateFactory.getInstance("X.509");
    final InputStream inputStream = new FileInputStream(file);
    final X509Certificate certificate = (X509Certificate)
certificateFactoryX509.generateCertificate(inputStream);
    inputStream.close();

    return certificate;
}

private static PrivateKey createPrivateKeyFromPemFile(final String keyFileName)
throws IOException, InvalidKeySpecException, NoSuchAlgorithmException
{
    final PemReader pemReader = new PemReader(new FileReader(keyFileName));
    final PemObject pemObject = pemReader.readPemObject();
    final byte[] pemContent = pemObject.getContent();
    pemReader.close();
    final PKCS8EncodedKeySpec encodedKeySpec = new
PKCS8EncodedKeySpec(pemContent);
    final KeyFactory keyFactory = getKeyFactoryInstance();
    final PrivateKey privateKey = keyFactory.generatePrivate(encodedKeySpec);
    return privateKey;
}

```

The `createX509CertificateFromFile` method receives the certificate file name in the `certificateFileName` argument. First, the code creates a new `java.io.File` instance with the received certificate file name as an argument. If the file exists, the next line calls the `CertificateFactory.getInstance` method with "X.509" as an argument to generate a certificate factory object that implements the X.509 certificate type and saves its reference in the `certificateFactoryX509` variable. Then, the code calls the `generateCertificate` method with the file input stream to generate and return an `X509Certificate` instance from the received certificate file name.

The `createPrivateKeyFromPemFile` method receives the key file name in the `keyFileName` argument. First, the code creates a new Bouncy Castle `PemReader` instance to load the key file specified in the `keyFileName` argument with this reader capable of loading the PEM format. The code generates a byte array with the PEM content and saves it in the `pemContent` variable. The next line passes `pemContent` as an argument to create a new instance of the `PKCS8EncodedKeySpec` class and saves it in the `encodedKeySpec` variable. Then, the code calls the `getKeyFactoryInstance` static method defined in the same class that provides a `java.key.KeyFactoryinstance` that allows the code to convert public/private keys of the RSA algorithm, and saves the instance in the `keyFactory` variable. The call to the `keyFactory.generatePrivate` method with `encodedKeySpec` as an argument generates a private

key object from the provided key specification. Finally, the code returns this generated private key object.

The following lines declare the `createKeyManagerFactory` and `createTrustManagerFactory` methods within the `SecurityHelper` class. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01/src/SecurityHelper.java` file:

```
private static KeyManagerFactory createKeyManagerFactory(
    final String clientCertificateFileName, final String
clientKeyFileName, final String clientKeyPassword)
    throws InvalidKeySpecException, NoSuchAlgorithmException,
KeyStoreException, IOException, CertificateException, UnrecoverableKeyException
{
    final X509Certificate clientCertificate =
createX509CertificateFromFile(clientCertificateFileName);
    final PrivateKey privateKey =
createPrivateKeyFromPemFile(clientKeyFileName);
    final KeyStore keyStore =
KeyStore.getInstance(KeyStore.getDefaultType());
    keyStore.load(null, null);
    keyStore.setCertificateEntry("certificate", clientCertificate);
    keyStore.setKeyEntry("private-key", privateKey,
        clientKeyPassword.toCharArray(),
        new Certificate[] { clientCertificate });
    final KeyManagerFactory keyManagerFactory =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
    keyManagerFactory.init(keyStore, clientKeyPassword.toCharArray());

    return keyManagerFactory;
}

private static TrustManagerFactory createTrustManagerFactory(
    final String caCertificateFileName)
    throws CertificateException, NoSuchAlgorithmException, IOException,
KeyStoreException
{
    final X509Certificate caCertificate = (X509Certificate)
createX509CertificateFromFile(caCertificateFileName);
    final KeyStore keyStore =
KeyStore.getInstance(KeyStore.getDefaultType());
    keyStore.load(null, null);
    keyStore.setCertificateEntry("ca-certificate", caCertificate);
    final TrustManagerFactory trustManagerFactory =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
    trustManagerFactory.init(keyStore);

    return trustManagerFactory;
}
```

The `createKeyManagerFactory` method receives the client certificate file name in the `clientCertificateFileName` argument, the client key file name in the `clientKeyFileName` argument, and the password for the client key in the `clientKeyPassword` argument. First, the code calls the previously explained `createX509CertificateFromFile` method with `clientCertificateFileName` as an argument to load the client certificate, generate the `X509Certificate` instance, and save its reference in the `clientCertificate` variable.

The next line calls the previously explained `createPrivateKeyFromPemFile` method with `clientKeyFileName` as an argument to load the private client key with PEM format, generate the `PrivateKey` instance, and save its reference in the `privateKey` variable.

Then, the code creates an instance of the `KeyStore` class named `keyStore`, whose goal is to provide storage facility for cryptographic keys and certificates. The call to the `keyStore.setCertificateEntry` method assigns the trusted certificate saved in the `clientCertificate` variable to the "certificate" alias. The call to the `keyStore.setKeyEntry` method assigns the key saved in the `privateKey` variable to the "private-key" alias, sets its association with the certificate saved in `clientCertificate`, and protects it with the password received in the `clientKeyPassword` argument.

Finally, the code creates an instance of the `KeyManagerFactory` class and then calls its `init` method with `keyStore` and `clientKeyPassword` converted to a char array as the arguments to provide the `KeyStore` instance and the password to the key manager factory. The code returns the created and initialized key manager factory.

The `createTrustManagerFactory` method receives the certificate authority certificate file name in the `caCertificateFileName` argument. First, the code calls the previously explained `createX509CertificateFromFile` method with `caCertificateFileName` as an argument to load the certificate authority certificate, generate the `X509Certificate` instance, and save its reference in the `caCertificate` variable.

The next line creates an instance of the `KeyStore` class named `keyStore`, whose goal is to provide storage facility for cryptographic keys and certificates. The call to the `keyStore.setCertificateEntry` method assigns the trusted certificate authority certificate saved in the `caCertificate` variable to the "ca-certificate" alias.

Finally, the code creates an instance of the `TrustManagerFactory` class and then calls its `init` method with `keyStore` as an argument to provide the `KeyStore` instance and to the trust manager factory. The code returns the created and initialized trust manager factory.

The following lines complete the declaration of the `SecurityHelper` class and declare the public static `createSocketFactory` method. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01/src/SecurityHelper.java` file:

```
public static SSLSocketFactory createSocketFactory()
public static SSLSocketFactory createSocketFactory(
    final String caCertificateFileName,
    final String clientCertificateFileName,
    final String clientKeyFileName) throws Exception
{
    final String clientKeyPassword = "";
    try
    {
        Security.addProvider(new BouncyCastleProvider());
        final KeyManager[] keyManagers =
createKeyManagerFactory(clientCertificateFileName, clientKeyFileName,
clientKeyPassword).getKeyManagers();
```

```
        final TrustManager[] trustManagers =
createTrustManagerFactory(caCertificateFileName).getTrustManagers();

        final SSLContext context =
SSLContext.getInstance(TLS_VERSION);

        context.init(keyManagers, trustManagers, new
SecureRandom());

        return context.getSocketFactory();
    }
    catch (Exception e)
    {
        throw new Exception("I cannot create the TLS socket
factory.", e);
    }
}
```

The `createSocketFactory` method receives the certificate authority certificate file name in the `caCertificateFileName` argument, the client certificate file name in the `clientCertificateFileName` argument, and the client key file name in the `clientKeyFileName` argument. The method declares the `clientKeyPassword` string with an empty string as its value because we haven't used a password for the client key. In case you want to use a password, you can add the password as an argument to this helper method. Then, the code calls the `Security.addProvider` method with a new `BouncyCastleProvider` instance as an argument to add this instance to the security providers.

The next line calls the previously explained `createKeyManagerFactory` method with the appropriate arguments to generate the `KeyManager` instance. The result is chained with a call to the `getKeyManagers` method to include this instance in the new `keyManagers` array of `KeyManager`.

The next line calls the previously explained `createTrustManagerFactory` method with the appropriate argument to generate the `TrustManager` instance. The result is chained with a call to the `getTrustManagers` method to include this instance in the new `trustManagers` array of `TrustManager`.

Then, the code creates the TLS socket factory for the desired TLS version, that is, TLS version 1.2. Remember that we create an `SSLContext` instance named `context` but we are working with TLS. The call to `context.init` with `keyManagers`, `trustManagers`, and a new `java.security.SecureRandom` instance initializes the TLS socket factory with the previously generated key manager and trust manager. Finally, the code returns the result of calling the `context.getSocketFactory` method to return an `SSLSocketFactory` instance.

Configuring a TLS secured connection in Java

First, we will use the asynchronous API provided by the Paho Java client to create an MQTT asynchronous client that connects to the Mosquitto MQTT server. We will write a few lines of Java code to establish a secured connection, subscribe to a topic, and display the payload for the messages received in this topic. The synchronous API provided by the Paho Java client is simpler to use, but in most cases, we will want to use the non-blocking asynchronous API. Hence, we will focus on the asynchronous API.

In [Chapter 2, Securing an MQTT Mosquitto Server](#), we have secured our Mosquitto server, and therefore, we will use the digital certificates we have created to authenticate the client. Most of the times, we will work with an MQTT server that uses TLS, and therefore, it is a good idea to learn how to establish a connection with TLS and TLS authentication. It is easier to establish an unsecured connection with an MQTT server, but it won't be the most common scenario we will face when developing applications that work with MQTT.

First, we need to copy the following files we have created in [Chapter 2, Securing an MQTT Mosquitto Server](#), to a directory in the computer or device we will use to run the Java application. We saved the files in a directory called certificates. Create a `java_certificates` directory in the computer or board you are going to use as the MQTT client for this example. Copy the following three files to this new directory:

- `ca.crt`: Certificate authority certificate file
- `device001.crt`: Client certificate file
- `device001.key`: Client key

Now, we will create the main class named `Main` that uses the asynchronous API of the Paho Java Client and the previously coded `SecurityHelper` class to establish an encrypted connection with an MQTT server. Then, the class subscribes to the `commands/lights/led01` topic filter and prints all the messages received in the subscribed topic filter.

We will split the code for the `Main` class into many code snippets to make it easier to understand each code section. The following lines declare all the necessary imports. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01/src/Main.java` file:

```
import javax.net.ssl.SSLSocketFactory;  
  
import javax.net.ssl.SSLSocketFactory;  
  
import org.eclipse.paho.client.mqttv3.IMqttActionListener;  
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;  
import org.eclipse.paho.client.mqttv3.IMqttToken;  
import org.eclipse.paho.client.mqttv3.MqttAsyncClient;  
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;  
import org.eclipse.paho.client.mqttv3.MqttException;  
import org.eclipse.paho.client.mqttv3.MqttMessage;  
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
```

The following lines start the declaration of the `Main` class and show the first lines of the static `main`

method. You have to replace the value for the `mqttServerHost` with the IP address for the Mosquitto server or any other MQTT server you might decide to use. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01/src/Main.java` file:

```
public class Main {
    public static void main(String[] args) {
        try {
            final MqttConnectOptions mqttConnectOptions = new
MqttConnectOptions();
            mqttConnectOptions.setMqttVersion(
                MqttConnectOptions.MQTT_VERSION_3_1_1);
            mqttConnectOptions.setKeepAliveInterval(
                MqttConnectOptions.KEEP_ALIVE_INTERVAL_DEFAULT);
            MemoryPersistence memoryPersistence = new
MemoryPersistence();

            // Replace localhost with the IP for the Mosquitto
            // or other MQTT server
            final String mqttServerHost = "localhost";

            final int mqttServerPort = 8883;
            final String mqttServerURI = String.format(
                "ssl://%s:%d",
                mqttServerHost,
                mqttServerPort);

            // In case you don't need security:
            // final int mqttServerPort = 8883;
            // final String mqttServerURI = String.format(
            //     "ftp://%s:%d",
            //     mqttServerHost,
            //     mqttServerPort);

            // We can specify our own client id
            // instead of calling the generateClientId method
            final String mqttClientId =
MqttAsyncClient.generateClientId();
            MqttAsyncClient mqttAsyncClient = new MqttAsyncClient(
                mqttServerURI,
                mqttClientId,
                memoryPersistence);
            final String topic_for_led01 = "commands/lights/led01";
```

First, the code creates an instance of the `org.eclipse.paho.client.mqttv3.MqttConnectOptions` class named `mqttConnectOptions` that stores the values for the different options we can configure when we want to connect with an MQTT server. If we used the default parameters, we would work with MQTT version 3.1. We want to work with MQTT version 3.1.1, and therefore, we specify `MqttConnectOptions.MQTT_VERSION_3_1_1` as an argument for `mqttConnectOption.setMqttVersion`.

We want to work with MQTT version 3.1.1 interval configuration option. However, we wrote code to call the `mqttConnectOptions.setKeepAliveInterval` method to make it easy to work with other values in case we need to. We just need to change the value sent as an argument to this method.

Tip

You can call other methods that start with the `set` prefix for `mqttConnectOptions` to set the values for other configuration options, such as specifying a user name and a password for authentication. In this case, we are just working with TLS client authentication.

The next line creates an instance of the

`org.eclipse.paho.client.mqttv3.persist.MemoryPersistence` class named `memoryPersistence`. The Paho Java Client provides a pluggable persistence mechanism that makes it possible for the client to store the messages. In this case, we will work with a QoS level of 0, and we will use memory persistence for the messages to keep the code simpler. However, we will move to a QoS level of 2 later, and we will explore and configure other pluggable persistence options that will be suitable for this QoS level. All the pluggable persistence mechanisms are classes that implement the `MqttClientPersistence` interface.

Then, the code declares the values for the variables that specify the MQTT server host name or IP, `mqttServerHost`, the MQTT server port, `mqttServerPort`, and the MQTT server URI, `mqttServerURI`, built with the values of the other two variables. The next line calls the `MqttAsyncClient.generateClientId` method to generate a unique client ID and save it in the `mqttclientId` variable. We might also decide to specify our own client ID instead of calling this method.

The next line creates an instance of the `org.eclipse.paho.client.mqttv3.MqttAsyncClient` class named `mqttAsyncClient` with the MQTT server URI, the previously generated client ID, and the memory persistence option as the arguments. This instance will be the entry point for our MQTT asynchronous client.

Finally, the last line declares the `topic_for_led01` variable with "commands/lights/led01" as its value. We will subscribe to this topic.

The following lines continue the declaration of the public static `main` method. You have to replace the value, `/Users/gaston/certificates`, in the `certificatesPath` string with the path to the `java_certificates` directory you created. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01/src/Main.java` file:

```
// Replace /Users/gaston/certificates with the path
        // in which you saved the certificate authority file,
        // the client certificate file and the client key
        final String certificatesPath =
"/Users/gaston/certificates";
        final String caCertificateFileName =
String.join(java.io.File.separator, certificatesPath, "ca.crt");
        final String clientCertificateFileName =
String.join(java.io.File.separator, certificatesPath, "device001.crt");
        final String clientKeyFileName =
String.join(java.io.File.separator, certificatesPath, "device001.key");
        // In case you don't need security,
        // you don't need to create an SSLSocketFactory instance
```

```

SSLocketFactory socketFactory;
try {
    socketFactory = SecurityHelper.createSocketFactory(
        caCertificateFileName,
        clientCertificateFileName,
        clientKeyFileName);
} catch (Exception e1) {
    e1.printStackTrace();
    return;
}
// In case you don't need security, you don't have to call
// the setSocketFactory method
mqttConnectOptions.setSocketFactory(socketFactory);

```

The code declares the `certificatesPath` variable initialized with a string with the path in which you saved the certificate authority file, the client certificate file, and the client key. The code declares the following string variables with the full path to the certificate and key files we need to configure TLS and TLS client authentication: `caCertificateFileName`, `clientCertificateFileName`, and `clientKey`. The call to `String.join` with `java.io.File.separator` as the first argument makes it easy to join the path specified in the `certificatesPath` variable with each filename and to generate the full path. The `java.io.File.separator` string provides the appropriate name separator in the platform, and therefore, we don't have to worry about whether to use a slash (/) or a backslash () to join the path with the file name. Whenever we work with IoT related projects, sometimes, we can develop and test in Windows and then run the code on an IoT board that can use different Unix flavors. In other cases, we can develop on macOS or Linux and then run the code in a board that uses Windows IoT Core. The usage of `java.io.File.separator` makes our job easier in these scenarios, in which we switch between different platforms.

Then, the code calls the previously analyzed `SecurityHelper.createSocketFactory` method with the variables that hold the full path to the certificate and key files as arguments. This way, the code generates the `SSLocketFactory` instance and saves its reference in the `socketFactory` variable. Finally, the next line calls the `mqttConnectOptions.setSocketFactory` method with `socketFactory` as an argument to specify the socket factory we want to use for the connection with the MQTT server.

Working with asynchronous notifications and callbacks in Java

The following lines continue the declaration of the public static `main` method. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01/src/Main.java` file:

```
mqttAsyncClient.setCallback(new MqttCallback() {
    @Override
    public void connectionLost(Throwable cause) {
        cause.printStackTrace();
    }

    @Override
    public void deliveryComplete(IMqttDeliveryToken token) {
    }

    @Override
    public void messageArrived(String topic,
MqttMessage message) throws Exception {
        if (!topic.equals(topic_for_led01)) {
            return;
        }
        String messageText =
            new String(message.getPayload(),
"UTF-8");
        System.out.println(
            String.format("Topic: %s. Payload:
%s",
                topic,
                messageText));
    }
});
```

The code calls the `mqttAsyncClient.setCallback` method with an anonymous subclass of the `MqttCallback` class as an argument. The `setCallback` method requires an argument of the `org.eclipse.paho.client.mqttv3.MqttCallback` type. The anonymous class that implements the `MqttCallback` interface provides implementations for the following three methods required by the interface:

- `connectionLost`: When the MQTT client loses the connection, this method will be executed.
- `messageArrived`: When a message has arrived from the MQTT server, this method will be executed. It is very important to understand that in the QoS levels that require an acknowledgement to the MQTT server, the MQTT server won't send this acknowledgement until the `messageArrived` method completes its execution. The method receives the topic for the message that has arrived in the `topic` argument of type `String`. In addition, the method receives an instance of the `org.eclipse.paho.client.mqttv3.MqttMessage` class in the `message` argument that encapsulates all the data related to the arrived message.

- `deliveryComplete`: When the delivery for a message has been completed and all the acknowledgements required by the QoS level have been received, this method will be executed. The method receives an instance of a class that implements the `org.eclipse.paho.client.mqttv3.IMqttDeliveryToken` interface in the `token` argument. We will discuss tokens later.

Note

We must call the `setCallback` method before we establish the connection with the MQTT server. In this case, we pass an anonymous class as an argument to the `setCallback` method because we want all the code to be included within the `main` method. In other cases, we can create a class that implements the required interface and pass an instance of this class as an argument to the `setCallback` method.

In this case, the code just prints the stack trace for the cause that made the MQTT client lose the connection. We don't execute any code when the delivery for a message has been completed.

The `messageArrived` method checks whether the `topic` for the received message matches the value saved in the `topic_for_led01` variable. In case they match, the next line creates a new `String` with the byte array returned by the `message.getPayload` method and `UTF-8` as the encoding, and saves it in the `messageText` variable. Finally, the code prints the topic and the payload for the message.

The following lines end the public static `main` method and the `Main` class. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java01/src/Main.java` file:

```
// In this case, we don't use the token
    IMqttToken mqttConnectToken = mqttAsyncClient.connect(
        mqttConnectOptions,
        null,
        new IMqttActionListener() {
            @Override
            public void onSuccess(IMqttToken
asyncActionToken) {
                System.out.println(
                    String.format(
                        "Successfully
connected"));
            }
            try {
                // In this case, we
                IMqttToken
subscribeToken = mqttAsyncClient.subscribe(
                    topic_for_led01,
                    0,
                    null,
                    new
IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken asyncActionToken) {
```

The code calls the `mqttAsyncClient.connect` method with `mqttConnectOptions` and an anonymous class that implements the `IMqttActionListener` interface as the value for the `callback` argument. The `connect` method requires the `callback` argument of the `org.eclipse.paho.client.mqttv3.IMqttActionListener` type. The anonymous class provides implementations for the following two methods required by the `IMqttActionListener` interface:

- **onSuccess**: When the MQTT client has successfully completed the action, this method will be executed. The method receives an instance of a class that implements the `org.eclipse.paho.client.mqttv3.IMqttToken` interface in the `asyncActionToken` argument. We can compare this instance with the `IMqttToken` returned by the method that starts the asynchronous action to use the same code to check for the successful execution of many asynchronous actions. In this case, the `onSuccess` method will be executed when the connection has been successfully established with the MQTT server.

- **onFailure:** When an asynchronous operation requested to the MQTT client fails, this method will be executed. As happens in the `onSuccess` method, the `onFailure` method also receives an `asyncActionToken` argument with the `IMqttToken` instance, which identifies the asynchronous operation that has failed.

Subscribing to topics and understanding notification events in Java

The `onSuccess` method prints a message indicating that the client has successfully connected with the MQTT server and then calls the `mqttAsyncClient.subscribe` method with `topic_for_led01` as the argument that specifies the topic filter to subscribe to 0 for the argument that specifies the desired QoS level, `null` for the `userContext` argument, and another anonymous class that implements the `IMqttActionListener` interface as the value for the `callback` argument. As happened with the `connect` method, the `subscribe` method requires the `callback` argument of the `org.eclipse.paho.client.mqttv3.IMqttActionListener` type. The anonymous class provides implementations for the previously explained two methods required by the `IMqttActionListener` interface: `onSuccess` and `onFailure`.

The code in the `onSuccess` method prints a message indicating that the client has subscribed to the topic filter and the granted QoS level. The `asyncActionToken.getGrantedQos()` method returns an array of `int` that provides the QoS level that the MQTT server has granted for each of the topic filter subscription requests.

Tip

In this case, we pass an anonymous class as an argument to both the `connect` and `subscribe` methods because we want all the code to be included within the `main` method. In other cases, we can create a class that implements the required interface. It is also possible to take advantage of the tokens to use the same code base to process successful and failed executions for different asynchronous operations with the same code base. The token indicates the specific asynchronous operation that has fired the method. In the sample code we have written, we didn't check the tokens because we used a specific anonymous class for each asynchronous operation.

Make sure the Mosquitto server or any other MQTT server you might want to use for this example is running. Use your favorite Java IDE to run the example on any computer, or build a `.jar` file with all the dependencies included and launch it on any computer or device that you want to use as the MQTT client, with a command similar to the following line. In case you don't run the example on the same computer on which you are running the MQTT server, remember to change the value for the `mqttServerHost` string in the `main` method of the `Main` class. You must also copy the certificate files and the `.jar` file to the device. The next line uses `Sensors01-0.0.1-SNAPSHOT.jar` as the jar file for the previous example: **java -jarSensors01-0.0.1-SNAPSHOT.jar**

In case you see a stack trace with a `javax.net.ssl.SSLHandshakeException` exception as a result, make sure you check that the MQTT server hostname or IP matches the value specified for the `Common Name` attribute when you generated the server certificate file named `server.crt`. Make sure that you check the IP address for the MQTT server (Mosquitto server) and generate the server certificate file and key again with the appropriate IP address specified for `Common Name`, as explained in [Chapter 2, Securing an MQTT Mosquitto Server](#), in case you are working with the self-signed certificates we

generated in the aforementioned chapter.

Now, follow these steps to use the MQTT.fx GUI utility to publish two messages to the commands/lights/led01 topic:

1. Launch MQTT.fx and establish a connection with the MQTT server by following the steps we learned in [Chapter 2](#), *Securing an MQTT Mosquitto Server*.
2. Click **Publish** and enter commands/lights/led01 in the dropdown on the left-hand side of the **Publish** button.
3. Click **QoS 0** on the right-hand side of the **Publish** button.
4. Enter the following text in the textbox below the **Publish** button: TURN ON. Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic.
5. Enter the following text in the textbox below the **Publish** button: TURN OFF. Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic.

In case you don't want to work with the MQTT.fx utility, you can run two `mosquitto_pub` commands to generate MQTT clients that publish messages to the topic. You just need to open another Terminal in macOS or Linux, or another Command Prompt in Windows, go to the directory in which Mosquitto is installed, and run the following commands. Replace `localhost` with the IP for the MQTT server. Remember to replace `ca.crt`, `device001.crt`, and `device001.key` with the full path to these files created in the `java_certificates` directory: `mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt --key device001.key -t test/drone01 -m 'TURN ON' -q 0 --tls-version tlsv1.2 -d`

```
mosquitto_pub -h localhost -V mqttv311 -p 8883 --cafile ca.crt --cert device001.crt  
--key device001.key -t test/drone01 -m 'TURN OFF' -q 0 --tls-version tlsv1.2 -d
```

Go to the device and window in which you executed the Java application. You will see the following output: **Successfully connected Subscribed to the commands/lights/led01 topic with QoS: 0 Topic: commands/lights/led01. Payload: TURN ON Topic: commands/lights/led01. Payload: TURN OFF**

The Java application successfully established a secure and encrypted connection with the MQTT server and became a subscriber to the commands/lights/led01 topic with a granted QoS level of 0. The program displayed the two messages it has received in the commands/lights/led01 topic.

Press **Ctrl+C** to stop the execution of the program or use the appropriate command in case you are running the Java application in your favorite IDE. The generated MQTT client will close the connection with the MQTT server.

Processing commands that interact with sensors in Java

Now that you have understood how to work with the Paho Java Client, its non-blocking API, and its callbacks, we can develop the Java application that will process the commands that interact with sensors, based on the previously explained requirements.

We can start another Maven project in our favorite IDE and make the same edits to the `pom.xml` file that we did in our previous example to use the Paho Java Client with the Bouncy Castle libraries. In addition, we must include the previously coded `SecurityHelper` class. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java02` folder.

Now, we will create a new class named `SensorsManager` that implements the previously analyzed `MqttCallback` interface, specifically the `org.eclipse.paho.client.mqttv3.MqttCallback` interface. The class will process the messages that the MQTT client receives, provide many methods that will process the commands included in these messages, and simulate the retrieval of values from two sensors.

The `SensorsManager` class will provide the following methods:

- `publishMessage`: This method publishes the payload specified as a string to the indicated topic with a QoS level and a retained flag value. The method uses an UTF-8 encoding to transform the string into a bytes array, uses the asynchronous client to publish the message, and passes the received `IMqttActionListener` instance as the callback for events related to the publication of the message. We will use this method to make it easier to publish messages to different topics with diverse QoS levels.
- `publishProcessedCommandMessage`: This method publishes a message to the appropriate topic, indicating that a command has been successfully processed. The method ends up calling the previously explained `publishMessage` method to publish the message.
- `messageArrived`: When a message has arrived from the MQTT server, this method will be executed. This method processes the `TURN ON` and `TURN OFF` commands, and based on these commands and the topics in which the commands were received, the code changes the value of the flags that determine whether the sensors must publish data every 1 second or not.
- `loop`: This method checks the Boolean values for the flags that determine whether the sensors must publish data every 1 second or not. In order to keep the example simple, the code generates pseudo-random values to simulate that it is retrieving data from the sensors and, based on the values of the flags, publishes messages with the data for the sensors in the different topics that we described when we analyzed the requirements for the application.

We will split the code for the `SensorsManager` class into many code snippets to make it easier to understand each code section. The following lines declare all the necessary imports. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java02/src/SensorsManager.java` file: `package com.packt.mqtteesentials.Sensors02;`

```

import org.eclipse.paho.client.mqttv3.IMqttActionListener; import
org.eclipse.paho.client.mqttv3.IMqttDeliveryToken; import
org.eclipse.paho.client.mqttv3.MqttAsyncClient; import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttException; import
org.eclipse.paho.client.mqttv3.MqttMessage;

import java.io.UnsupportedEncodingException; import java.util.concurrent.ThreadLocalRandom;

The following lines start the declaration of the SensorsManager class. The code declares the many string
constants, private fields, and the two volatile Boolean flags that indicate whether the sunlight sensor is
turned on and whether the earth humidity sensor is turned on: isSunlightSensorTurnedOn and
isEarthHumiditySensorTurnedOn. The code also declares the constructor for this class. The code file
for the sample is included in the mqtt_essentials_gaston_hillar_04 folder, in the
Java01/src/SensorsManager.java file: public class SensorsManager implements MqttCallback {
private static final String SENSOR_EARTH_HUMIDITY = "earthhumidity"; private static final String
SENSOR_SUNLIGHT = "sunlight"; private static final String TOPIC_SEPARATOR = "/"; private final
String boardCommandsTopic; private final String boardDataBaseTopic; private final String encoding;

private final MqttAsyncClient asyncClient; private final String earthHumidityTopic; private final String
visibleLightTopic; private final String infraredLightTopic; private final String ultraVioletIndexTopic;
private volatile boolean isSunlightSensorTurnedOn = false; private volatile boolean
isEarthHumiditySensorTurnedOn = false; public SensorsManager(final MqttAsyncClient asyncClient,
final String boardCommandsTopic, final String boardDataBaseTopic, final String encoding) {
this.boardCommandsTopic = boardCommandsTopic; this.boardDataBaseTopic = boardDataBaseTopic;
this.encoding = encoding;

this.asyncClient = asyncClient;

// Build and save the topic names that we will use to publish the data from the sensors
this.earthHumidityTopic = this.boardDataBaseTopic.concat(SENSOR_EARTH_HUMIDITY); final String
sunlightDataBaseTopic = boardDataBaseTopic.concat(SENSOR_SUNLIGHT); this.visibleLightTopic =
String.join(TOPIC_SEPARATOR, sunlightDataBaseTopic, "visiblelight"); this.infraredLightTopic =
String.join(TOPIC_SEPARATOR, sunlightDataBaseTopic, "ir"); this.ultraVioletIndexTopic =
String.join(TOPIC_SEPARATOR, sunlightDataBaseTopic, "uv"); }

```

The constructor receives the asynchronous client (`asyncClient`), the topic that we will use for the commands (`boardCommandsTopic`), the prefix for the topic that we will use to publish the data for the sensors (`boardDataBaseTopic`), and the encoding we are using for strings in the payload (`encoding`). The constructor saves all these received arguments in fields with the same names. Then, the constructor builds and saves the topic names that we will use to publish the data from the sensors.

The following lines declare the `publishMessage` and `publishProcessedCommandMessage` methods within the `SensorsManager` class. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the Java02/src/SensorsManager.java file: **public**

IMqttDeliveryToken publishMessage(final String topic, final String textForMessage,

IMqttActionListener actionListener, final int qos, final boolean retained) {

```
byte[] bytesForPayload;

try {
    bytesForPayload = textForMessage.getBytes(this.encoding); return asyncClient.publish(topic,
bytesForPayload, qos, retained, null, actionListener); } catch (UnsupportedEncodingException e) {
    e.printStackTrace();
    return null;
} catch (MqttException e) {
    e.printStackTrace();
    return null;
}
```

```
public void publishProcessedCommandMessage(final String sensorName, final String command) {
    final String topic = String.format("%s/%s", boardCommandsTopic, sensorName); final String
textForMessage = String.format(
    "%s successfully processed command: %s", sensorName, command); publishMessage(topic,
textForMessage, null, 0, false); }
```

The following lines declare the three methods required by the `MqttCallback` interface within the `SensorsManager` class: `connectionLost`, `deliveryComplete` and `messageArrived`. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java02/src/SensorsManager.java` file:

```
@Override
```

```
public void connectionLost(Throwable cause) {
    cause.printStackTrace();
}

@Override
public void deliveryComplete(IMqttDeliveryToken token) {
}
```

```
@Override

public void messageArrived(String topic, MqttMessage message) throws Exception {
    String messageText = new String(message.getPayload(), encoding); System.out.println(
        String.format("Topic: %s. Payload: %s", topic,
        messageText));

    // A message has arrived from the MQTT broker // The MQTT broker doesn't send back // an
    acknowledgment to the server until // this method returns cleanly

    if (!topic.startsWith(boardCommandsTopic)) {

        // The topic for the arrived message doesn't start with boardTopic return;
    }

    final boolean isTurnOnMessage = messageText.equals("TURN ON"); final boolean isTurnOffMessage =
    messageText.equals("TURN OFF"); boolean isInvalidCommand = false;

    boolean isInvalidTopic = false;

    // Extract the sensor name from the topic String sensorName = topic.replaceFirst(boardCommandsTopic,
    "").replaceFirst(TOPIC_SEPARATOR, ""); switch (sensorName) {

        case SENSOR_SUNLIGHT:

            if (isTurnOnMessage) {

                isSunlightSensorTurnedOn = true; } else if (isTurnOffMessage) {

                isSunlightSensorTurnedOn = false; } else {

                    isInvalidCommand = true;
                }
            break;

        case SENSOR_EARTH_HUMIDITY:

            if (isTurnOnMessage) {

                isEarthHumiditySensorTurnedOn = true; } else if (isTurnOffMessage) {
```

```

isEarthHumiditySensorTurnedOn = false; } else {

isValidCommand = true;

}

break;

default:

isValidTopic = true;

}

if (!isValidCommand && !isValidTopic) {

publishProcessedCommandMessage(sensorName, messageText); }

}

```

The following lines declare the `loop` method and finish the declaration of the `SensorsManager` class. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java02/src/SensorsManager.java` file:

```
public void loop() {
```

```

if (isEarthHumiditySensorTurnedOn) {

// Retrieve the humidity level from the sensor // In this case, we just generate a random number final int
humidityLevel = ThreadLocalRandom.current().nextInt(1, 101); // Publish the message to the appropriate
topic publishMessage(earthHumidityTopic, String.format("%d %%", humidityLevel), null, 0, false); }

if (isSunlightSensorTurnedOn) {

// Retrieve the visible light level from the sensor // In this case, we just generate a random number final int
visibleLight = ThreadLocalRandom.current().nextInt(201, 301); // Publish the message to the
appropriate topic publishMessage(visibleLightTopic, String.format("%d lm", visibleLight), null, 0,
false); }

// Retrieve the infrared light level from the sensor // In this case, we just generate a random number final int
infraredLight = ThreadLocalRandom.current().nextInt(251, 281); // Publish the message to the
appropriate topic publishMessage(infraredLightTopic, String.format("%d lm", infraredLight), null, 0,
false); }

// Retrieve the ultraviolet (UV) index from the sensor // In this case, we just generate a random number final int
ultraVioletIndex = ThreadLocalRandom.current().nextInt(0, 16); // Publish the message to the
appropriate topic publishMessage(ultraVioletIndexTopic, String.format("%d UV Index",
ultraVioletIndex), null, 0, false); }

}
```

}

Working with last will and testament in Java

Now, we will take advantage of the last will and testament feature. We will create the main class named `Main` that uses the asynchronous API of the Paho Java Client, the previously coded `SecurityHelper` class to establish an encrypted connection with an MQTT server, and the `SensorsManager` class to process the received messages with commands. The code will configure a message to be sent as the last will when the connection is lost. In addition, the `Main` class will call the `loop` method for a `SensorsManager` instance to publish messages with data retrieved from the sensors when they are turned on with the appropriate commands.

We will split the code for the `Main` class into many code snippets to make it easier to understand each code section. The following lines declare all the necessary imports. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java02/src/Main.java` file:

```
package com.packt.mqtteesentials.Sensors02;

import java.io.UnsupportedEncodingException;

import javax.net.ssl.SSLSocketFactory;

import org.eclipse.paho.client.mqttv3.IMqttActionListener;
import org.eclipse.paho.client.mqttv3.IMqttToken;
import org.eclipse.paho.client.mqttv3.MqttAsyncClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.persist.MqttDefaultFilePersistence;
```

The following lines start the declaration of the `Main` class and show the first lines of the static `main` method. You have to replace the value for the `mqttServerHost` with the IP address for the Mosquitto server or any other MQTT server you might decide to use. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java02/src/Main.java` file:

```
public class Main {
public class Main {
    private final static String BOARD_NAME = "location001";
    private final static String ENCODING_FOR_PAYLOAD = "UTF-8";

    public static void main(String[] args) {
        final String boardCommandsTopic =
String.format("commands/boards/%s", BOARD_NAME);
        final String boardDataBaseTopic = String.format("data/boards/%s/", BOARD_NAME);
        final String boardStatusTopic = String.format("status/boards/%s", BOARD_NAME);
        final String commandsTopicFilter = String.format("%s/+",
boardCommandsTopic);

        try {
            final MqttConnectOptions mqttConnectOptions = new
MqttConnectOptions();
            mqttConnectOptions.setMqttVersion(
```

```

        MqttConnectOptions.MQTT_VERSION_3_1_1);
        mqttConnectOptions.setKeepAliveInterval(
            MqttConnectOptions.KEEP_ALIVE_INTERVAL_DEFAULT);

        // Last will message configuration
        final String lastWillMessageText = "OFFLINE";
        byte[] bytesForPayload;
        try {
            bytesForPayload =
lastWillMessageText.getBytes(ENCODING_FOR_PAYLOAD);
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
            return;
        }
        mqttConnectOptions.setWill(boardStatusTopic,
bytesForPayload, 2, true);

        MqttDefaultFilePersistence filePersistence = new
MqttDefaultFilePersistence();

        // Replace localhost with the IP for the Mosquitto server
        // or other MQTT server
        final String mqttServerHost = "localhost";

        final int mqttServerPort = 8883;
        final String mqttServerURI = String.format(
            "ssl://%s:%d",
            mqttServerHost,
            mqttServerPort);

        // In case you don't need security:
        // final int mqttServerPort = 8883;
        // final String mqttServerURI = String.format(
        //     "ftp://%s:%d",
        //     mqttServerHost,
        //     mqttServerPort);

        // We can specify our own client id
        // instead of calling the generateClientId method
        //final String mqttClientId =
MqttAsyncClient.generateClientId();
        final String mqttClientId = BOARD_NAME;
        MqttAsyncClient mqttAsyncClient = new MqttAsyncClient(
            mqttServerURI,
            mqttClientId,
            filePersistence);

```

First, the code builds many strings with the topics that we will use to subscribe and publish messages. Then, the code creates an instance of the `org.eclipse.paho.client.mqttv3.MqttConnectOptions` class named `mqttConnectOptions`, which stores the values for the different options we can configure when we want to connect with an MQTT broker. If we used the default parameters, we would work with MQTT version 3.1. We want to work with MQTT version 3.1.1, and therefore, we specify `MqttConnectOptions.MQTT_VERSION_3_1_1` as an argument for `mqttConnectOption.setMqttVersion`.

Then, the code configures the last will message by calling the `mqttConnectOptions.setWill` method with the topic, the bytes for the payload, the QoS level set to 2, and `true` as the value for the retained flag. This way, if the client connection is interrupted or the Java application stops running, the configured last will message will be sent to the specified topic indicating that the board is offline.

The next line creates an instance of the

`org.eclipse.paho.client.mqttv3.persist.MqttDefaultFilePersistence` class named `filePersistence`. In this case, we want to use file persistence to store the messages because we will publish some messages with a QoS level of 2. We use the default directory for the file that persists the messages. You can explore the different configurations based on your needs.

Then, the code declares the values for the variables that specify the MQTT server host name or IP, `mqttServerHost`, the MQTT server port, `mqttServerPort`, and the MQTT server URI, `mqttServerURI`, built with the values of the other two variables. The next line sets the board name as the value for the `mqttclientId` variable. This way, we specify a fixed value for our client ID.

The next line creates an instance of the `org.eclipse.paho.client.mqttv3.MqttAsyncClient` class named `mqttAsyncClient` with the MQTT server URI, the previously specified client ID, and the file persistence option as the arguments. This instance will be the entry point for our MQTT asynchronous client.

The following lines continue the declaration of the public static `main` method. You have to replace the value, `/Users/gaston/certificates`, in the `certificatesPath` string with the path to the `java_certificates` directory you created. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_04` folder, in the `Java02/src/Main.java` file:

```
// Replace /Users/gaston/certificates with the path
        // in which you saved the certificate authority file,
        // the client certificate file and the client key
        final String certificatesPath =
"/Users/gaston/certificates";
        final String caCertificateFileName =
String.join(java.io.File.separator, certificatesPath, "ca.crt");
        final String clientCertificateFileName =
String.join(java.io.File.separator, certificatesPath, "device001.crt");
        final String clientKeyFileName =
String.join(java.io.File.separator, certificatesPath, "device001.key");
        // In case you don't need security,
        // you don't need to create an SSLSocketFactory instance
        SSLSocketFactory socketFactory;
try {
        socketFactory = SecurityHelper.createSocketFactory(
                caCertificateFileName,
                clientCertificateFileName,
                clientKeyFileName);
    } catch (Exception e1) {
        e1.printStackTrace();
        return;
    }
    // In case you don't need security, you don't have to call
    // the setSocketFactory method
```

```

        mqttConnectOptions.setSocketFactory(socketFactory);

        SensorsManager sensorsManager = new
SensorsManager(mqttAsyncClient,
                           boardCommandsTopic, boardDataBaseTopic,
ENCODING_FOR_PAYLOAD);

        mqttAsyncClient.setCallback(sensorsManager);

```

The first lines of code are similar to the code we analyzed in the previous example. The last lines create an instance of the SensorsManager class named sensorsManager and then call the mqttAsyncClient.setCallback method with this instance as an argument.

The following lines end the public static main method and the Main class. The code file for the sample is included in the mqtt_essentials_gaston_hillar_04 folder, in the Java02/src/Main.java file:

```

// In this case, we don't use the token
        IMqttToken mqttConnectToken = mqttAsyncClient.connect(
            mqttConnectOptions,
            null,
            new IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken
asyncActionToken) {
                    System.out.println(
                        String.format(
                            "Successfully
connected"));
                try {
                    // In this case, we
                    IMqttToken
subscribeToken = mqttAsyncClient.subscribe(
commandsTopicFilter,
                                2,
                                null,
                                new
IMqttActionListener() {
                    @Override
                    public void onSuccess(IMqttToken asyncActionToken) {
                        System.out.println(
                            String.format(
                                "Subscribed to the %s topic with QoS: %d",
                                asyncActionToken.getTopics()[0],
                                asyncActionToken.getGrantedQos()[0]));
                    }
                    // Publish a retained message indicating the board is online

```

```
sensorsManager.publishMessage(boardStatusTopic, "ONLINE", null, 2, true);
}

@Override
public void onFailure(IMqttToken asyncActionToken,
Throwable exception) {
exception.printStackTrace();
}
});

} catch (MqttException e) {
e.printStackTrace();
}

}

@Override
public void onFailure(IMqttToken
asyncActionToken,
Throwable exception) {
// This method is fired when an
exception.printStackTrace();
}

});

while (true) {
sensorsManager.loop();
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
System.err.format(
"Sleep interruption: %s",
e.toString());
}
}
} catch (MqttException e) {
e.printStackTrace();
}
}
```

The code calls the `mqttAsyncClient.connect` method with `mqttConnectOptions` and an anonymous class that implements the `IMqttActionListener` interface as the value for the `callback` argument. The `connect` method requires the `callback` argument of the `org.eclipse.paho.client.mqttv3.IMqttActionListener` type. The anonymous class provides implementations for the following two methods required by the `IMqttActionListener` interface: `onSuccess` and `onFailure`.

The `onSuccess` method prints a message indicating that the client has successfully connected with the

MQTT server and then calls the `mqttAsyncClient.subscribe` method with `commandsTopicFilter` as the argument, which specifies the topic filter to subscribe to, `2` for the argument that specifies the desired QoS level, `null` for the `userContext` argument, and another anonymous class that implements the `IMqttActionListener` interface as the value for the `callback` argument. The anonymous class provides implementations for the previously explained two methods required by the `IMqttActionListener` interface: `onSuccess` and `onFailure`.

The code in the `onSuccess` method prints a message indicating that the client has subscribed to the topic filter and the granted QoS level. Then, the code calls the `sensorsManager.publish` method to publish a retained message, indicating that the board is online with a QoS level of `2`.

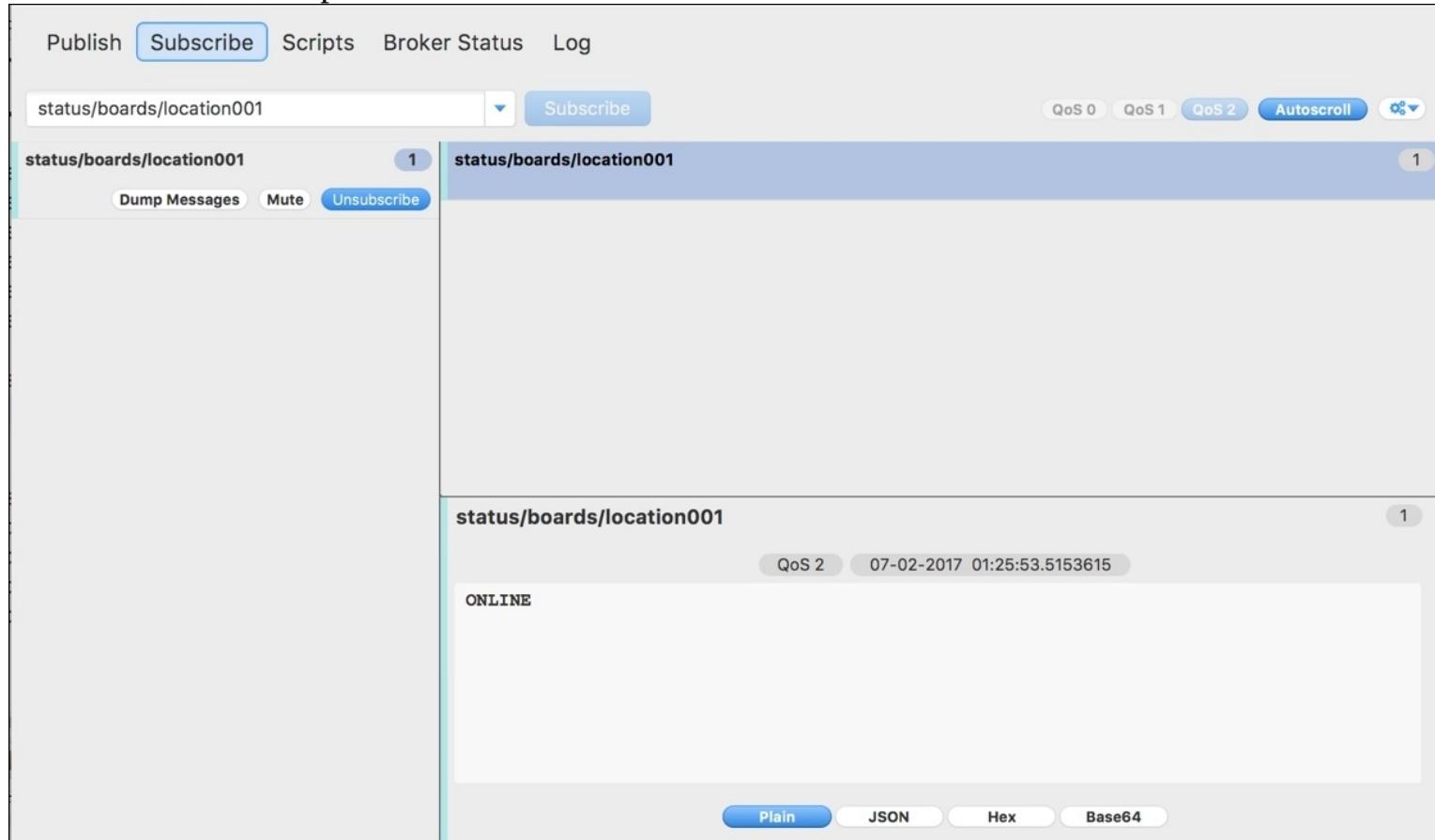
After the code calls the `mqttAsyncClient.connect` method, the method runs a `while` loop forever. The loop calls the `sensorsManager.loop` method to check the values for the flags that determine whether the method must publish the values for the different sensors to the appropriate topics or not. Then, the loop calls the `Thread.sleep` method to make the thread sleep for `1` second.

Publishing messages with commands to target sensors

Make sure the Mosquitto server or any other MQTT server you might want to use for this example is running. Use your favorite Java IDE to run the example in any computer or build a .jar file with all the dependencies included and launch it in any computer or device that you want to use as the MQTT client with a command similar to the following line. In case you don't run the example in the same computer in which you are running the MQTT server, remember to change the value for the `mqttServerHost` string in the `main` method of the `Main` class. You must also copy the certificate files and the .jar file to the device. The next line uses `Sensors02-0.0.1-SNAPSHOT.jar` as the JAR file for the previous example: `java -jar Sensors02-0.0.1-SNAPSHOT.jar`

Now, follow these steps to use the MQTT.fx GUI utility to subscribe to many topics and publish messages with commands to two topics to control the sensors.

1. Launch MQTT.fx and establish a connection with the MQTT server by following the steps you learned in [Chapter 2, Securing an MQTT Mosquitto Server](#).
2. Click **Subscribe**, click **QoS 2**, and enter `status/boards/location001` in the dropdown on the left-hand side of the **Subscribe** button. Then, click the **Subscribe** button. MQTT.fx will display a new panel on the left-hand side with the topic filter to which we have subscribed with QoS level 2 and a message with `ONLINE` as a payload. We subscribed after the code that published this message was executed. However, the message appears in our subscription because it was published with the retained flag set to `true`. The following screenshot shows the message displayed in MQTT.fx after we subscribe to the topic:



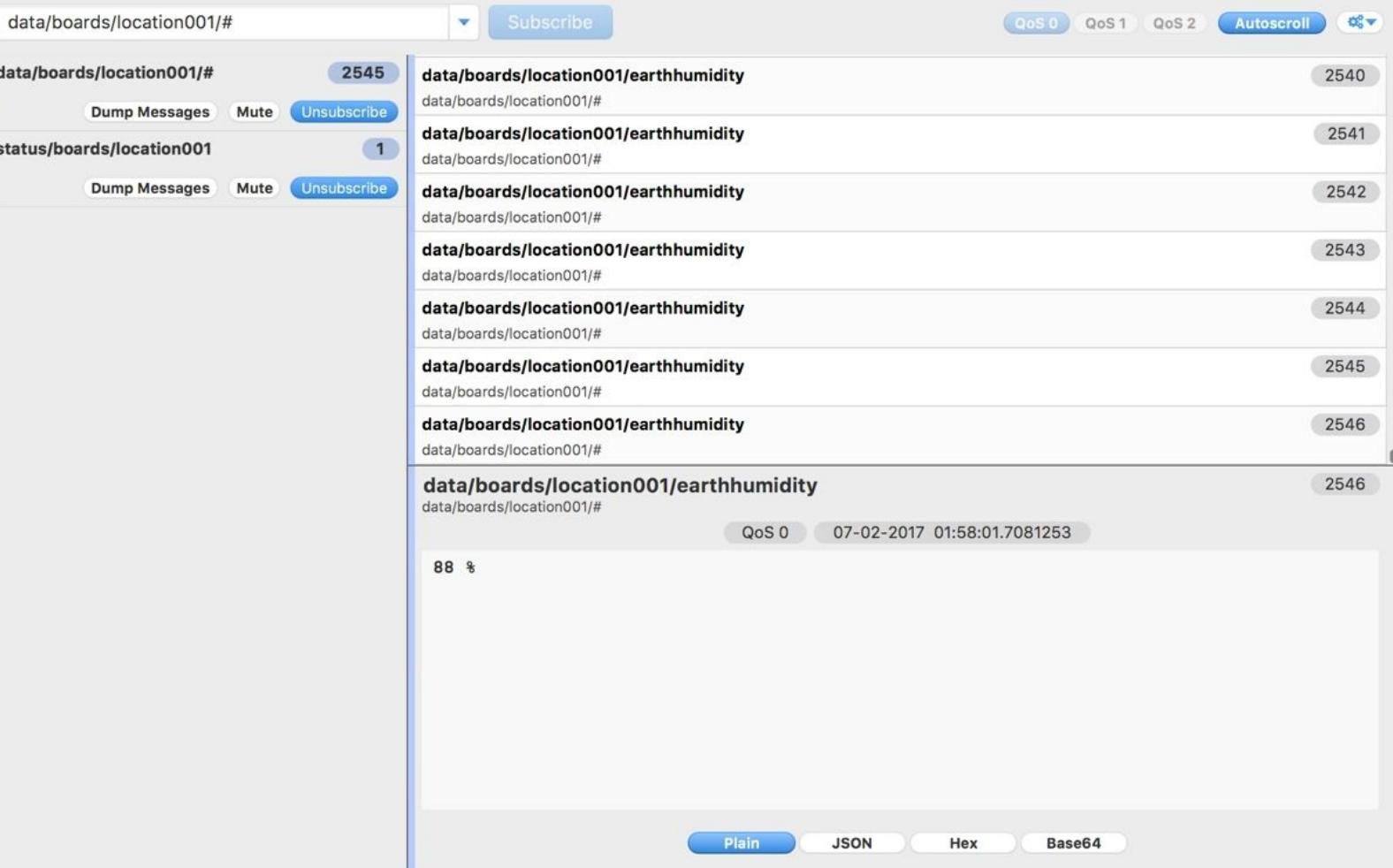
- Enter data/boards/location001/# in the dropdown on the left-hand side of the **Subscribe** button. Then, click **QoS 0** and the **Subscribe** button. MQTT.fx will display a new panel on the left-hand side with the topic filter to which we have subscribed with QoS level 0. No messages will appear that match this topic filter because the Java application has the two sensors turned off and they aren't publishing data.
- Click **Publish** and enter commands/boards/location001/sunlight in the dropdown on the left-hand side of the **Publish** button.
- Click **QoS 2** on the right-hand side of the **Publish** button.
- Enter the following text in the textbox below the **Publish** button: TURN ON. Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic. This way, the Java application will turn on the sunlight sensor and it will start publishing data to the data/boards/location001/sunlight/uv, data/boards/location001/sunlight/visiblelight, and data/boards/location001/sunlight/ir topics.
- Enter commands/boards/location001/earthhumidity in the dropdown on the left-hand side of the **Publish** button.
- Enter the following text in the textbox below the **Publish** button: TURN ON. Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic. This way, the Java application will turn on the sunlight sensor and it will start publishing data to the data/boards/location001/earthhumidity topic.
- Click **Subscribe** and you will see many messages that match the data/boards/location001/# topic filter. The following screenshot shows many of the messages received that match the topic filter after we turned on the sensors through commands in messages:

The screenshot shows the MQTT.fx interface with the 'Subscribe' tab selected. In the center pane, a list of received messages is displayed. The messages are as follows:

- data/boards/location001/sunlight/visiblelight (QoS 0, 1674)
- data/boards/location001/sunlight/ir (QoS 0, 1675)
- data/boards/location001/sunlight/uv (QoS 0, 1676)
- data/boards/location001/earthhumidity (QoS 0, 1677)
- data/boards/location001/sunlight/visiblelight (QoS 0, 1678)
- data/boards/location001/sunlight/ir (QoS 0, 1679)
- data/boards/location001/sunlight/uv (QoS 0, 1680)
- data/boards/location001/sunlight/uv** (QoS 0, 1680)
data/boards/location001/#
5 UV Index

At the bottom of the central pane, there are four buttons: Plain, JSON, Hex, and Base64. The 'Plain' button is highlighted.

- Click **Publish** and enter commands/boards/location001/sunlight in the dropdown on the left-hand side of the **Publish** button.
- Enter the following text in the textbox below the **Publish** button: TURN OFF. Then, click the **Publish** button. MQTT.fx will publish the entered text to the specified topic. This way, the Java application will turn off the sunlight sensor and it will stop publishing data to the data/boards/location001/sunlight/uv, data/boards/location001/sunlight/visiblelight, and data/boards/location001/sunlight/ir topics.
- Click **Subscribe** and you will see that only the messages for the data/boards/location001/earthhumidity topic are arriving and matching the data/boards/location001/# topic filter. The following screenshot shows many of the messages received that match the topic filter after we turned off the sunlight sensor through a command in a message:



Go to the device and the window in which you executed the Java application. You will see the following output: **Successfully connected Subscribed to the commands/boards/location001/+ topic with QoS: 2 Topic: commands/boards/location001/sunlight. Payload: TURN ON Topic: commands/boards/location001/sunlight. Payload: sunlight successfully processed command: TURN ON Topic: commands/boards/location001/earthhumidity. Payload: earthhumidity successfully processed command: TURN ON Topic: commands/boards/location001/earthhumidity. Payload: earthhumidity successfully processed command: TURN OFF Topic: commands/boards/location001/sunlight. Payload: sunlight successfully processed command: TURN OFF**

Press ***Ctrl+C*** to stop the execution of the program or use the appropriate command in case you are running the Java application in your favorite IDE. The generated MQTT client will close the connection with the MQTT server and the configured last will message will be sent to the **status/boards/location001** topic.

Go back to MQTT.fx and click **Subscribe**. You will see that a message with **OFFLINE** as the payload has arrived to the **status/boards/location001** topic. The following screenshot shows the configured last will message that has arrived in MQTT.fx:

status/boards/location001

[Subscribe](#)

QoS 0 QoS 1 QoS 2 Autoscroll

data/boards/location001/# 2683

[Dump Messages](#) [Mute](#) [Unsubscribe](#)

status/boards/location001 2

[Dump Messages](#) [Mute](#) [Unsubscribe](#)

data/boards/location001/earthhumidity 2679

data/boards/location001/#

data/boards/location001/earthhumidity 2680

data/boards/location001/#

data/boards/location001/earthhumidity 2681

data/boards/location001/#

data/boards/location001/earthhumidity 2682

data/boards/location001/#

data/boards/location001/earthhumidity 2683

data/boards/location001/#

data/boards/location001/earthhumidity 2684

data/boards/location001/#

status/boards/location001 2685

status/boards/location001 2685

QoS 2 07-02-2017 02:00:20.7220346

OFFLINE

[Plain](#)[JSON](#)[Hex](#)[Base64](#)

Test your knowledge

1. The Eclipse Paho Java Client provides:
 1. Only a blocking (synchronous) API.
 2. Only a non-blocking (asynchronous) API.
 3. Both a blocking (synchronous) and a non-blocking (asynchronous) API.
2. In the Paho Java API, an instance of `MqttAsyncClient` represents:
 1. The synchronous MQTT client.
 2. The asynchronous MQTT client.
 3. The asynchronous MQTT message.
3. After the `MqttAsyncClient` instance establishes a connection with the MQTT server, which of the following methods declared in the `IMqttActionListener` interface will be called?
 1. `onSuccess`.
 2. `connectCallback`.
 3. `onConnect`.
4. After the `MqttAsyncClient` instance receives a message from one of the topic filters to which it has subscribed, which of the following methods declared in the `MqttCallback` interface will be called?
 1. `onSuccess`.
 2. `onMessage`.
 3. `messageArrived`.
5. Which of the following methods of an `MqttConnectOptions` instance allows you to set a last will message?
 1. `setLastWillMessage`.
 2. `setLastWill`.
 3. `setWill`.

Summary

In this chapter, we analyzed the requirements to control sensors wired to different IoT boards with MQTT messages. We defined the topics that we would use and the commands that would be part of the messages' payloads to control sensors. We decided to take advantage of the last will and testament feature of MQTT. Then, we worked with the Eclipse Paho Java Client to write Java code that connected a non-blocking (asynchronous) MQTT client to the MQTT server.

We created a security helper class to establish a TLS secured connection and we worked with asynchronous notifications and callbacks in Java. We subscribed to topics, understood how notification events worked, and we wrote code that processed commands that interacted with sensors.

Finally, we worked with a GUI utility to interact with our Java application through MQTT messages. We were able to turn the sensors on and off and retrieve data from them.

Now that we have understood how to use Java to work with MQTT, we will use MQTT over WebSockets to control home automation with an HTML 5 web page, JavaScript, and Node.js, which is what we are going to discuss in the next chapter.

Chapter 5. Controlling Home Automation in JavaScript

In this chapter, we will write JavaScript code to control home automation with MQTT messages. We will write code that will be able to run on web browsers. In addition, we will use Node.js to write code that runs on IoT boards, receives MQTT messages and controls devices wired to the boards. We will:

- Understand the requirements to control home automation with MQTT and WebSockets
- Define the topics and commands to control home automation
- Enable WebSockets for a Mosquitto server
- Understand the Eclipse Paho JavaScript asynchronous client
- Create an HTML 5 web page to control LEDs
- Publish messages to target remote devices in JavaScript
- Process the received messages in JavaScript
- Work with WebSockets to connect to the MQTT server
- Subscribe to topics in JavaScript
- Process commands that interact with LEDs in Node.js
- Run the home automation web application
- Debug the home automation web application
- Secure MQTT over WebSockets with TLS

Understanding the requirements

Now, we want to be able to control home automation in any web browser. Hence, we have to be able to send and receive MQTT messages in these web browsers. There are no W3C standards for implementing MQTT support in web browsers. No major web browser has implemented direct MQTT support.

Luckily, MQTT over WebSockets makes it possible for us to encapsulate MQTT messages on WebSockets and use any modern web browser that supports WebSockets as a client for an MQTT server that also provides support for WebSockets.

Tip

Mosquitto provides support for MQTT over WebSockets. However, we must enable this support by making changes in the configuration file. In case you don't want to work with Mosquitto, you must make sure that the MQTT server you use provides support for MQTT over WebSockets to work with this example.

First, we will create a web page with HTML 5 and JavaScript code to generate an MQTT client that will act as a publisher and subscriber. We will connect a JavaScript MQTT client to the MQTT server via WebSockets, and we will generate commands to control lights with MQTT messages.

These commands will allow us to change the colors of many lights wired to different IoT boards that will run JavaScript Node.js code to subscribe to topics and process the messages with the commands. In this case, we will connect a Node.js MQTT client to the MQTT server via WebSockets, and we will process the received commands. We want our JavaScript code to run on many platforms because we will use the same code base to change the colors of lights that use the following IoT boards. It is possible to install and run Node.js in all of these boards:

- Raspberry Pi 3
- Intel Edison
- Intel Joule 570x
- Intel Joule 550x

Each platform will work with different kinds of lights and other elements that we will control as part of our home automation solution. However, we will focus on the basic features that will allow us to change the colors of **LEDs (Light Emitting Diodes)** to keep our example simple and to stay concentrated on MQTT over WebSockets. Then, we will be able to use this project as a baseline for other solutions that require us to run code on an IoT board that runs JavaScript Node.js code, has to connect to an MQTT server via WebSockets, and process commands.

The code that runs on the boards that have LEDs wired to them must be able to process commands received in messages in a specific topic. We will use JSON strings in the payloads.

The client application, that is, the web page, must allow us to select the desired color for different LEDs that might be connected to different boards. The client application doesn't need to know the board to which each LED is wired. The JavaScript code in the web page will publish MQTT messages with JSON strings to the topic for each LED. The client application must display the results of executing each

command. Each IoT board must publish a message to a specific topic whenever a command is successfully executed.

Defining the topics and commands to control home automation

We will use the following topic name to publish the commands for an LED:

`home/control/leds/lednumber`, where **lednumber** must be replaced with a unique numeric ID assigned to an LED wired to a board. For example, if we assign 1 as the ID for an LED that is wired to an Intel Joule 570x board, we will have to publish commands to the `home/control/leds/1` topic and the code that runs on this board will subscribe to this topic to receive messages with commands and react to them.

Note that the topic name doesn't include details about the board and just focuses on the ID assigned to the LED. This is a specific requirement because it is necessary to make it possible to change the board to which the LEDs are wired without requiring edits in the client code. The client code doesn't have to know anything about the boards. Hence, the code that runs on the IoT boards must subscribe to all the topics for the different LEDs that they have wired. Of course, everything we have learned so far about MQTT, that is, topics and filters, will provide you with many ideas on how to organize the topics. In this case, we are just using one possible solution that is definitely not the best one but will allow us to learn how MQTT works over WebSockets. The solution has been chosen for simplicity rather than scalability.

We will use the following topic name to make the IoT boards publish details about the successfully executed commands: `home/results/leds/lednumber`, where **lednumber** must be replaced with a unique numeric ID assigned to an LED wired to the board. For example, if we assign 5 as the ID for an LED that is wired to a Raspberry Pi 3 board, the client that wants to receive information about successfully processed commands has to subscribe to the `home/results/leds/5` topic.

The commands will be sent in a JSON string with a key-value pair. Our example will support just one command that allows us to set the color for an LED. The key will always be "Color" and the value will be the desired RGB (short for Red Green Blue) color as a hexadecimal string.

The following line shows an example of the payload for a command that makes an LED display the red color:

```
{"Color": "FF0000"}
```

The following line shows an example of the payload for a command that makes an LED display the blue color:

```
{"Color": "0000FF"}
```

We have all the necessary details to start coding with HTML 5, JavaScript, and Node.js.

Enabling WebSockets for a Mosquitto server

Now, we will configure Mosquitto to work with WebSockets. First, we will work with WebSockets without TLS, and therefore, the communications between the client and Mosquitto through WebSockets will be unsecure. This way, any client that knows the host name and port will be able to establish a communication with the Mosquitto server.

In case you are running the Mosquitto server in a Terminal window in macOS or Linux, press *Ctrl+C* to stop it. In Windows, stop the appropriate service.

Go to the Mosquitto installation directory and open the `mosquitto.conf` configuration file. In [Chapter 2, Securing an MQTT Mosquitto Server](#), we learned that it is a good practice to make a backup copy of the existing `mosquitto.conf` configuration file before making changes to it. This way, we can easily roll back to the previous configuration if something goes wrong.

In macOS, Linux, or Windows, add the following lines at the end of the configuration file:

```
listener 9001 0.0.0.0      protocol websockets
```

We specified the `9001` value for the `listener` option to make Mosquitto listen for incoming network connections on port `9001`. This port is a very common port number for MQTT over WebSockets.

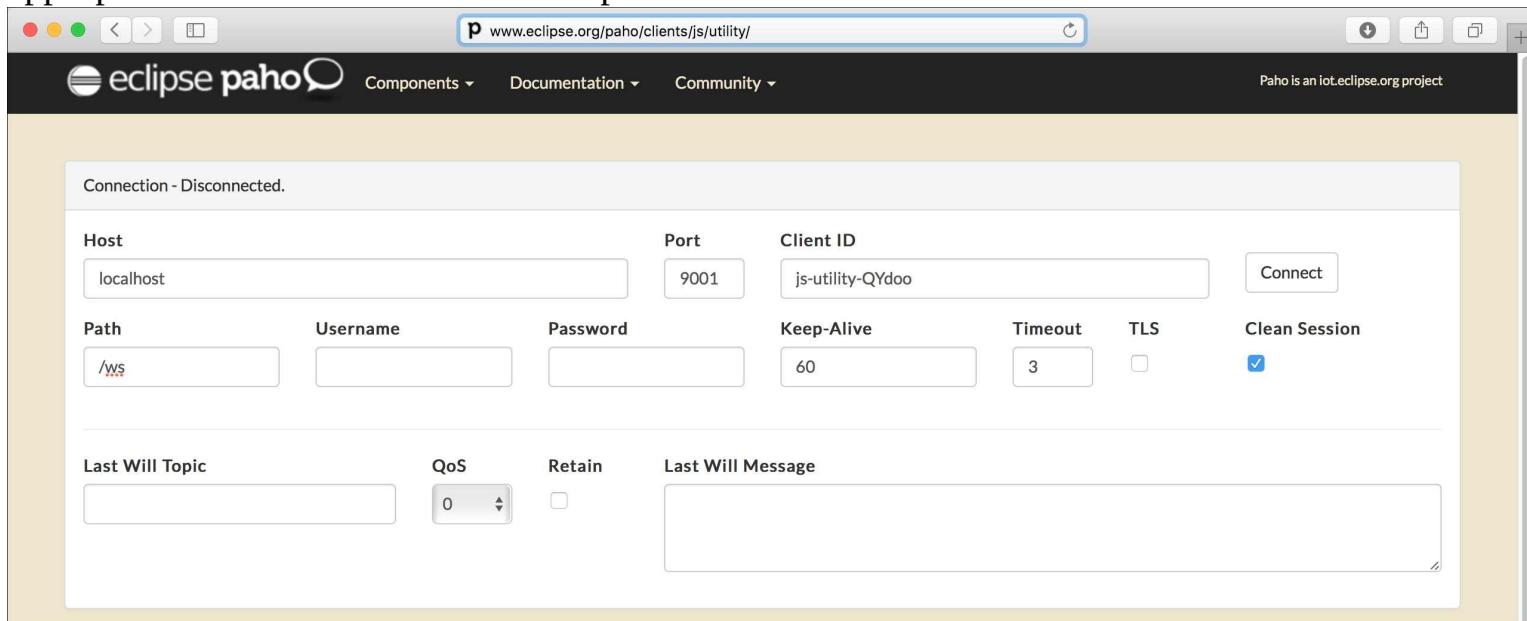
The `protocol` option is the next line after we defined the new listener and specified `websockets` as the desired protocol to use on port `9001`. This way, we enable Mosquitto's support for MQTT over WebSockets on port `9001`.

Save the changes to the `mosquitto.conf` configuration file and launch Mosquitto again with the same mechanism we learned for Linux, macOS, and Windows in [Chapter 1, Lightweight Messaging with MQTT 3.1.1 and Mosquitto](#). The Mosquitto server will add WebSockets support on port `9001` to the previously configured listener that will still work with MQTT with TLS on port `8883`. It is very important to understand that the certificates we configured for the listener that work on port `8883` do not apply to the WebSockets listener on port `9001`. Each listener has its own TLS configuration. In this case, we aren't using TLS for WebSockets on port `9001`.

Testing WebSockets with the Eclipse Paho client JavaScript utility

We will use the Eclipse Paho client JavaScript utility to generate a simple MQTT over WebSockets client on a web browser. We will make this client subscribe to a topic and print all the messages it receives. Follow the next steps:

1. Open a web browser in the same computer in which you are running the Mosquitto server and navigate to <http://www.eclipse.org/paho/clients/js/utility>.
2. Enter localhost in **Host**.
3. Enter 9001 in **Port**.
4. Make sure /ws is specified in **Path**.
5. Make sure the **TLS** checkbox is unchecked. The following screenshot shows the web page with the appropriate values for the connection options:



- Click **Connect**. The web page will display **Connected to: localhost:9001/ws** as, followed by the Client ID that was automatically generated.
- Go to the **Subscribe** panel. Enter home/control/leds/1 in **Topic**, below **Subscribe**.
- Select 2 in the **QoS** dropdown below **Subscribe**. The following screenshot shows the **Subscribe** panel with the appropriate values for the subscription options:

Subscribe

Topic home/control/leds/1

QoS 2

Subscribe

Unsubscribe

- Click the **Subscribe** button.
- Go to the **Publish Message** panel. Enter home/control/leds/1 in **Topic**, below **Publish Message**.
- Select 2 in the **QoS** dropdown below **Publish Message**.
- Enter the following text in the **Message** textbox below **Publish Message**: {"Color": "FFA0A0"}.
- The following screenshot shows the **Publish Message** panel with the appropriate values for the message publication options:

Publish Message

Topic

home/control/leds/1

QoS

2

Retain



Publish

Message

{"Color": "FFA0A0"}

- Click the **Publish** button. The web page will use MQTT over WebSockets to publish the message. The web page is a subscriber to the topic to which we published the message, and therefore, we will see the message displayed in the **Last Messages** panel at the bottom of the page. The following screenshot shows the **Last Messages** panel with the message that has arrived and matched the subscription's topic filter we specified:

Last Messages

Topic	Payload	Time	QoS
home/control/leds/1	{"Color": "FFA0AO"}	2017-02-23T04:52:25.244Z	2

[History](#)

[Clear History](#)

Topic	Payload	Time	QoS
home/control/leds/1	{"Color": "FFA0AO"}	2017-02-23T04:52:25.244Z	2

With the configuration changes we have made to the Mosquitto server, any client that uses MQTT over WebSockets will be able to establish a connection with Mosquitto, and subscribe and publish to topics. Remember that with the configuration we are using, the data sent between the MQTT over WebSocket clients and the MQTT server isn't encrypted.

Understanding the Eclipse Paho JavaScript asynchronous client

In [Chapter 4, Interacting with Sensors in Java](#), you learned that the Eclipse Paho project provides an open source client implementation of MQTT, and you worked with its Java client. The project also includes a JavaScript client, also known as the Paho JavaScript client or Eclipse Paho JavaScript client. The web page for the Eclipse Paho project is <http://www.eclipse.org/paho>. The web page for the Eclipse Paho JavaScript client is <http://eclipse.org/paho/clients/js>. This JavaScript client provides only a non-blocking (asynchronous) API.

The previous web page uses the Eclipse Paho JavaScript client to allow us to subscribe to topic filters and publish messages to topics. Now, we will use the same JavaScript client in a new web page.

Tip

In this case, you will download the plain library. Take into account that you can also consume the plain library and the minified library from cdnjs.com, as explained in the previously mentioned web page for this library.

The JavaScript client exposes a single object name in the global namespace. The global `Paho.MQTT` module has a single public function named `client`. This function returns a `Paho.MQTT.client` object based on the specified connection details.

The Paho JavaScript API is composed of the following two main objects:

- `Paho.MQTT.Client`: This object represents the asynchronous MQTT over WebSockets client and allows us to specify the callbacks that notify the application when specific events occur, such as a new message arriving and a connection being established with the MQTT server.
- `Paho.MQTT.Message`: This object encapsulates an MQTT message with its topic and its payload. It is very important to understand that the Paho JavaScript API uses the destination name to refer to the topic for a message.

Creating an HTML 5 web page to control LEDs

First, we will create a new HTML 5 file called `home_automation.html`. The web page will also use a pure JavaScript color picker: JSColor. The main web page for JSColor is <http://jscolor.com>. The following lines show the code for the single web page for our web application. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_05` folder, in the `home_automation.html` file:

```
<html>
  <head>
    <script src="paho.javascript-1.0.2/mqttws31.js"></script>
    <script src="jscolor-2.0.4/jscolor.js"></script>
    <script src="home_automation.js"></script>
    <title>MQTT over WebSockets Sample</title>
  </head>

  <body>
    <div id="status">Not connected with MQTT Server</div>
    <div id="led-1">
      <span>
        Select color for LED #1
      </span>
      <input class="jscolor" onchange="APP.HomeAutomation.Manager.updateLed(1, this.jscolor)" value="#FFFFFF">
      <svg height="100" width="100">
        <circle id="status-circle-led-1" cx="50" cy="50" r="40" stroke="black" stroke-width="3" fill="#FFFFFF" />
      </svg>
      <div id="status-message-led-1">
      </div>
    </div>

    <div id="led-2">
      <span>
        Select color for LED #2
      </span>
      <input class="jscolor" onchange="APP.HomeAutomation.Manager.updateLed(2, this.jscolor)" value="#FFFFFF">
      <svg height="100" width="100">
        <circle id="status-circle-led-2" cx="50" cy="50" r="40" stroke="black" stroke-width="3" fill="#FFFFFF" />
      </svg>
      <div id="status-message-led-2">
      </div>
    </div>

    <div id="led-3">
      <span>
        Select color for LED #3
      </span>
      <input class="jscolor" onchange="APP.HomeAutomation.Manager.updateLed(3, this.jscolor)" value="#FFFFFF">
      <svg height="100" width="100">
        <circle id="status-circle-led-3" cx="50" cy="50" r="40" stroke="black" fill="#FFFFFF" />
      </svg>
      <div id="status-message-led-3">
      </div>
    </div>
  </body>

```

```

stroke-width="3" fill="#FFFFFF" />
    </svg>
    <div id="status-message-led-3">
        </div>
</div>

</body>
</html>

<script>

document.addEventListener("DOMContentLoaded", function(event) {
    APP.HomeAutomation.Manager.connect();
});

</script>

```

The first lines load the following three scripts:

- `mqttws31.js`: This is the Paho JavaScript client script file
- `jscolor.js`: This is the JSColor script file
- `home_automation.js`: This is a script file that we will create later and that will declare the `APP.HomeAutomation.Manager` object

The body section declares a `div` element whose `id` is `status` and will change its text when a connection is successfully established with the MQTT server.

Then, the code declares three `div` elements with almost the same code: `led-1`, `led-2`, and `led-3`. Each `div` element represents an LED. In this case, we repeated the code and we just replaced the LED number in the different parts of the code. However, in a real-life application, you will work with a framework that will allow you to use templates to generate these pieces of code where just one identifier changes. We want to keep things simple and use HTML 5 and JavaScript without any additional templates or frameworks.

Each `div` that represents an LED declares an `input` element with the `jscolor` class that will display the JSColor pure JavaScript color picker. When the user clicks or taps on this `input` element, a color picker will be displayed, and a click on a tap or a color will fire the code specified for `onchange`. The following line shows the code that runs for the first LED:

```
APP.HomeAutomation.Manager.updateLed(1, this.jscolor)
```

The line calls the `APP.HomeAutomation.Manager.updateLed` function with the LED identifier as an argument (1) and a string with the selected color from the color picker (`this.jscolor`).

In addition, the `div` that represents an LED declares an SVG circle whose `id` is `status-circle-led-`, followed by the LED identifier. Whenever the IoT board successfully updates the color for the LED and publishes a status message, the web page will receive this message and update the fill color for the circle with the color the user has chosen.

Finally, the `div` that represents an LED declares a `div` element whose `id` is `status-message-led-`,

followed by the LED identifier. The web page will change its text indicating the color that has been set for the LED in the IoT board.

The script element at the end of the code uses pure JavaScript to call the `APP.HomeAutomation.Manager.connect` function once the page DOM (short for Document Object Model) is ready for the JavaScript code to execute. In case you are working with jQuery, you can call this function within a `$(document).ready` block. However, we want to keep the example with pure JavaScript.

Note

In order to keep the example simple and focused on the code, we don't create a CSS file with styles for the different elements and/or classes declared in the HTML 5 file.

Publishing messages to target remote devices in JavaScript

Now, we will create a new JavaScript file named `home_automation.js` that will provide an `APP.HomeAutomation.Manager` object with many strings, objects, and functions that we will use to make it easy to establish a connection with MQTT over WebSockets. The functions will work with the Paho JavaScript client.

We will split the code for the `home_automation.js` file in many code snippets to make it easier for us to understand each code section. The following lines start the declaration of the `APP.HomeAutomation.Manager` object. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_05` folder, in the `home_automation.js` file:

```
var APP = APP || {};
var APP = APP || {};
APP.HomeAutomation = APP.HomeAutomation || {};
APP.HomeAutomation.Manager = {
    ledCommandBaseTopic: "home/leds/",
    ledResultBaseTopic: "home/results/leds/",

    // Replace with the host name for the MQTT Server
    host: "localhost",
    // Replace with the port number for MQTT over WebSockets the MQTT Server
    port: 9001,
    clientId: "home-web-" + clientId: "home-web-" +
Math.random().toString(16).substr(2, 8),

    mqttConnectOptions: {
        timeout: 3,
        // I want to use MQTT Version 3.1.1
        mqttVersion: 4,
        mqttVersionExplicit: true,
        // For TLS security, you just need to set useSSL to True
        // and install the necessary certificates
        // useSSL: true,
        useSSL: false,
        cleanSession: true
    },
}
```

The previous code declares many variables that we will use within the functions declared in the `APP.HomeAutomation.Manager` object. The `mqttConnectOptions` object specifies values for the different options that we will use to establish a connection with the MQTT Server. We will pass this object as an argument when we ask the Paho JavaScript client to connect with the MQTT server. We specify the necessary values to work with MQTT version 3.1.1, use a clean session, and work with an unsecured connection.

The following lines declare the `updateLed`, `onConnectionLost`, `onMessageArrived`, and `onMessageDelivered` functions within the `APP.HomeAutomation.Manager` object and `createTrustManagerFactory`. The code file for the sample is included in the

mqtt_essentials_gaston_hillar_05 folder, in the home_automation.js file:

```
updateLed: function(ledId, jscolor) {
updateLed: function(ledId, jscolor) {
    console.log('I will send ' + jscolor + ' to LED #' + ledId);
    var payload = {
        "Color": jscolor.toString()
    };

    payloadString = JSON.stringify(payload);
    message = new Paho.MQTT.Message(payloadString);
    message.destinationName = this.ledCommandBaseTopic + ledId;
    message.qos = 0;

    this.client.send(message);
},
onConnectionLost: function(responseObject) {
    if (responseObject.errorCode !== 0)
        console.log("onConnectionLost: " + responseObject.errorMessage);
},
onMessageArrived: function(message) {
    console.log("Message arrived for topic: " + message.destinationName + ", with the following payload: " + message.payloadString);
    if
(!message.destinationName.startsWith(APP.HomeAutomation.Manager.ledResultBaseTopic))
{
    return;
}
    var ledNumber =
message.destinationName.replace(APP.HomeAutomation.Manager.ledResultBaseTopic, "");
    var payload = JSON.parse(message.payloadString);
    if (ledNumber && payload.Color) {
        // Update the status for the LED that has successfully changed its color
        var statusLedDiv = document.getElementById("status-message-led-" + ledNumber);
        //statusLedDiv.style.backgroundColor = "#" + payload.Color;
        statusLedDiv.textContent = "LED's color set to #" + payload.Color;
        var statusLedCircle = document.getElementById("status-circle-led-" + ledNumber);
        statusLedCircle.style.fill = "#" + payload.Color;
    },
onMessageDelivered: function(message) {
},
```

The updateLed function receives the LED ID and the RGB color as a hexadecimal string in the ledId and jscolor arguments. The code prints a message to the console log indicating that it will send the color received in jscolor to the LED identifier received in the ledId argument. Then, the code creates a new object named payload with "Color" as the key and the result of jscolor.toString() as the value. The next line calls the JSON.stringify function with payload as an argument to generate the JSON string representation of the object and to save it in the payloadString variable.

Then, the code creates a new `Paho.MQTT.Message` instance named `message` with `payloadString` as an argument. Then, the code sets the value of the `message.destinationName` attribute to `this.ledCommandBaseTopic`, concatenated with the value received in the `ledId` argument. The `destinationName` attribute for a `Paho.MQTT.Message` instance specifies the topic to which a message has to be published.

The default value for the QoS level for a message is 0. However, the next line sets the value for the `message.qos` attribute to 0 to make it easy to change it if we want to work with other QoS levels.

The `connect` function that we will analyze later creates a `Paho.MQTT.Client` instance and saves it in `this.client`. Finally, the `updateLed` function calls the `this.client.send` function with the message as an argument to publish the MQTT message. The function runs with an asynchronous execution, and after the message is successfully delivered, the `Paho.MQTT.Client` instance will execute the callback function assigned to the `client.onMessageDelivered` attribute in the `connect` function. Remember that a delivered message has a different meaning according to the QoS level that we use to publish a message.

The `onConnectionLost` function receives a `responseObject` object as an argument. This function prints a message to the console log with the value of the `responseObject.errorMessage` attribute when the `responseObject.errorCode` attribute is not equal to 0.

Processing the received messages in JavaScript

The `onMessageArrived` function receives a `Paho.MQTT.Message` instance in the `message` argument. The code prints a message to the console log indicating that a message has arrived with the destination topic and the payload string included in the `message.destinationName` and `message.payloadString` attributes.

Then, the code checks whether `message.destinationName` starts with `APP.HomeAutomation.Manager.ledResultBaseTopic` to make sure that the message's destination topic is the one defined for the results of the commands related to LEDs.

Note

In this case, the code only subscribes to the topic filters that we defined for publishing messages with the results of the commands related to LEDs. However, we include this code to understand how we should check the `message.destinationName` value when we subscribe to multiple topics that have different goals.

Then, the code retrieves the LED ID from the message's topic and saves it in the `ledNumber` variable. The code removes `APP.HomeAutomation.Manager.ledResultBaseTopic` from `message.destinationName` to retrieve the LED ID. For example, if the value for `message.destinationName` is "home/results/leds/1", the code replaces "home/results/leds/" with an empty string and the resulting string, "1", is the LED ID.

The next line calls `JSON.parse` with `message.payloadString` as an argument to parse the message's payload string and create an object from the received JSON string. The line saves the resulting object in the `payload` variable. In this case, we don't validate the content of `message.payloadString` to make sure that it is a valid JSON in order to keep the example simple.

Then, the code checks whether `ledNumber` and `payload.color` have values. If they have values, the function uses pure JavaScript to update the status message and the circle's fill color for the LED that has successfully changed its color. The code retrieves the element whose ID is equal to "status-message-led" concatenated with `ledNumber` and saves it in the `statusLedDiv` variable. The next line changes the value of the `statusLedDiv.textContent` attribute to indicate that the LED's color has been set to the color indicated in `payload.color`. This way, the user will know that the color that he selected in the corresponding color picker in the web application has been set to the LED.

Finally, the code retrieves the element whose ID is equal to "status-circle-led" concatenated with `ledNumber` and saves it in the `statusLedCircle` variable. The next line assigns "#" concatenated with `payload.color` to the `statusLedCircle.style.fill` attribute to use the same color that has been set to the LED as the fill color for the circle that represents the LED in the web application.

The `onMessageDelivered` function receives a `Paho.MQTT.Message` instance in the `message` argument. In this case, the function has an empty body because we don't want to run code when a message has been successfully delivered.

Working with WebSockets to connect to the MQTT server

The following lines declare the `onConnectSuccess` and `connect` functions within the `APP.HomeAutomation.Manager` object. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_05` folder, in the `home_automation.js` file:

```
onConnectSuccess: function(invocationContext) {
onConnectSuccess: function(invocationContext) {
    // Update the status text
    document.getElementById("status").textContent = "Connected with the MQTT
Server";
    // Now, subscribe to home/results/leds/1, home/results/leds/2 and
    // home/results/leds/3
    var client = invocationContext.invocationContext.client;
    for (var i = 1; i < 4; i++) {
        client.subscribe("home/results/leds/" + i);
    }
},
connect: function() {
    this.client = new Paho.MQTT.Client(this.host, this.port, this.clientId);
    this.client.onConnectionLost = this.onConnectionLost;
    this.client.onMessageArrived = this.onMessageArrived;
    this.client.onMessageDelivered = this.onMessageDelivered;
    // I want to receive the client in the onSuccess and onFailure callbacks
    // Hence, I save a reference to this.client in invocationContext
    this.mqttConnectOptions.invocationContext = {
        client: this.client
    };
    this.mqttConnectOptions.onSuccess = this.onConnectSuccess;
    this.mqttConnectOptions.onFailure = function (message) {
        console.log("Connection has failed: " + message);
    }
    this.client.connect(this.mqttConnectOptions);
}
};
```

The `connect` function creates a new instance of `Paho.MQTT.Client` with `this.host`, `this.port`, and `this.clientId` as the arguments that specify the host, the port number, and the MQTT client ID. The code saves this new instance in the `this.client` attribute. It is very important to understand that creating an instance of `Paho.MQTT.Client` doesn't establish a connection with the MQTT server. The `Paho.MQTT.Client` instance saved in `this.client` represents an MQTT over the WebSockets client, and we will use it to communicate with an MQTT server.

When we declare a function, we pass this function as an argument to another function or method, or we assign this function to an attribute and then some code calls this function at some time. This mechanism is known as **callback**. The name **callback** is used because the code calls back a function at some time.

The previous code snippets declared three functions that we will specify as callbacks:

- `onConnectSuccess`: This function will be called when the MQTT client receives a CONNACK response from the MQTT server, that is, when a connection has been successfully established with the MQTT server.
- `onConnectionLost`: This function will be called when the MQTT client has lost connection with the MQTT server.
- `onMessageArrived`: This function will be called when the MQTT client receives a PUBLISH message from the MQTT server. Whenever the MQTT server publishes a message based on the subscriptions for the client, this function will be called.
- `onMessageDelivered`: This function will be called when the MQTT client successfully delivers a message.

Then, the code sets the values for the following attributes that specify the callbacks that we want to be executed when certain events occur: `onConnectionLost`, `onMessageArrived`, and `onMessageDelivered`. We want to receive `this.client` in the `onSuccess` and `onFailure` callbacks, and therefore, the code creates an object with `this.client` as the value for the `client` attribute and assigns this object to the `this.mqttConnectOptions.invocationContext` attribute. We will use `this.mqttConnectOptions` as an argument for the `this.client.connect` method and the value specified for the `invocationContext` attribute will be sent as an argument whenever the client calls the callbacks specified for `onSuccess` and `onFailure`.

The next two lines set the values for the `onSuccess` and `onFailure` attributes for `this.mqttConnectOptions`. Finally, the code calls the `this.client.connect` function with `this.mqttConnectOptions` as an argument. This way, the code asks the MQTT over the WebSockets client to establish a connection with the MQTT server specified in `this.mqttConnectOptions`.

Note

The `connect` function runs with an asynchronous execution, and therefore, it is a non-blocking call.

Subscribing to topics in JavaScript

After a connection has been successfully established with the MQTT server, the specified callback in the `this.mqttConnectOptions.onSuccess` attribute will be executed, that is, the `onConnectSuccess` function. This function receives an `invocationContext` object as an argument. The code can access the object we assigned to `this.mqttConnectOptions.invocationContext` in `invocationContext.invocationContext`. Thus, the code can retrieve the instance of `Paho.MQTT.Client`, which established the connection with the MQTT server in the `invocationContext.invocationContext.client` attribute.

First, the `onSuccess` function updates the status text to indicate that the client is connected with the MQTT server. Then, the code saves `Paho.MQTT.Client`, which established the connection in the `client` variable. Finally, a `for` loop calls the `client.subscribe` method three times to subscribe to the following topics: `home/results/leds/1`, `home/results/leds/2`, and `home/results/leds/3`.

Tip

The Paho JavaScript client allows us to subscribe to a single topic filter at a time, and therefore, we have to make three calls to `client.subscribe` with the desired topic filter as an argument.

In this case, we don't specify any subscribe options because we will work with the default QoS level of 0, and we don't want to specify any callbacks for the success or failure of the subscription. However, we must take into account that it is possible to specify an object with additional subscription options as a second argument to the `subscribe` function.

Processing commands that interact with LEDs in Node.js

Now that we have coded the web application that will allow the user to change the colors of the three LEDs, we will write the Node.js code that will control the LEDs wired to the IoT board. We can run the Node.js code in any computer or IoT board capable of running Node.js version 6.9.x or higher.

For this example, we have specified in the requirements that the LEDs would be wired to the following IoT boards: Intel Edison, Intel Joule 550x, Intel Joule 570x, and Raspberry Pi 3 boards. All of these boards can have Node.js installed and run the sample code. However, we can also run the code on many other IoT boards.

MQTT.js is an open source library that allows us to work with MQTT over WebSockets and direct MQTT with Node.js. The library is very easy to use and will allow us to process the received messages and publish the results by writing just a few lines of code. The web page for the MQTT.js library is <http://www.npmjs.com>.

Make sure you have Node.js installed in the computer or IoT board you will use to run this example. Then, run the following command to install MQTT.js. Note that the code is tested with MQTT.js 2.3.1. Based on your Node.js installation directory in macOS or Ubuntu Linux, you might need to run the next command with sudo as a prefix, that is, you might need to run sudo npm install mqtt. In Windows, you might need to run the next command in a Command Prompt executed as an Administrator:

```
npm install mqtt
```

You will see an output similar to the following lines. However, the results might be different based on the modules that you may have already installed:

```
mqtt@2.3.1 node_modules/mqtt
└── inherits@2.0.3
└── reinterval@1.1.0
└── xtend@4.0.1
└── minimist@1.2.0
└── split2@2.1.1 (through2@2.0.3)
└── concat-stream@1.6.0 (typedarray@0.0.6)
└── commist@1.0.0 (leven@1.0.2)
└── readable-stream@2.2.2 (buffer-shims@1.0.0, util-deprecate@1.0.2,
string_decoder@0.10.31, process-nextick-args@1.0.7, isarray@1.0.0, core-util-
is@1.0.2)
└── mqtt-packet@5.2.1 (bl@1.2.0, process-nextick-args@1.0.7)
└── end-of-stream@1.1.0 (once@1.3.3)
└── pump@1.0.2 (once@1.4.0)
└── websocket-stream@3.3.3 (through2@2.0.3, ws@1.1.2, duplexify@3.5.0)
└── help-me@1.0.1 (callback-stream@1.1.0, through2@2.0.3, glob-stream@5.3.5)
```

Now, we will create a new Node.js JavaScript file named leds_control.js. The code file for the sample is included in the mqtt_essentials_gaston_hillar_05 folder, in the leds_control.js file:

```

var mqtt = require("mqtt")

// Replace with the host name for the MQTT Server
var host = "localhost"
// Replace with the port number for MQTT over WebSockets the MQTT Server
var port = 9001
// If we want to work with TLS, we must use the next line
// because we must use wss:// instead of ws://
//var client = mqtt.connect("wss://" + host + ":" + port)
var client = mqtt.connect("ws://" + host + ":" + port)

var ledCommandBaseTopic = "home/leds/"
var ledResultBaseTopic = "home/results/leds/"

client.on("connect", function () {
  console.log("I'm connected to the MQTT server")
  var topicFilters = Array()
  for (i = 1; i < 4; i++) {
    topicFilters.push(ledCommandBaseTopic + i)
  }
  client.subscribe(topicFilters)
})

client.on('message', function (topic, message) {
  // message is Buffer
  var payloadString = message.toString()
  console.log("Message arrived for topic: " + topic + ", with the following
payload: " + payloadString)
  if (!topic.startsWith(ledCommandBaseTopic)) {
    return;
  }
  var ledNumber = topic.replace(ledCommandBaseTopic, "")
  var payload = JSON.parse(payloadString)
  if (ledNumber && payload.Color) {
    console.log("I've changed LED #" + ledNumber + " to " + payload.Color)
  }
  var resultMessagePayload = {
    "Color": payload.Color
  }
  resultMessagePayloadString = JSON.stringify(resultMessagePayload)

  client.publish(ledResultBaseTopic + ledNumber, resultMessagePayloadString)
})

```

First, the code requires `mqtt` and saves the object in the `mqtt` variable. Then, the code declares the `host` and `port` variables with the host name for the MQTT server and the port number for MQTT over WebSockets. Then, the code calls the `mqtt.connect` function with `ws://`, followed by the host value, a colon (:), and the port number. If we want to connect to localhost on port 9001, the value for this argument will be `ws://localhost:9001`. We specify `ws` as the protocol name because we want `MQTT.js` to create a WebSockets over MQTT connection. The results of this function are assigned to the `client` variable.

Then, the code specifies the callback function assigned to the `connect` event. This function will be called after the client successfully establishes a connection with the MQTT server.

Note

After the MQTT over WebSockets client establishes a connection with the MQTT server, `client.connected` will evaluate to `true`.

The function creates a `topicFilters` array with all the topics to which the client wants to subscribe to in order to control the three LEDs. Then, the code calls the `client.subscribe` function with `topicFilters` as an argument to subscribe to the following three topics with a single subscription request: `home/leds/1`, `home/leds/2`, and `home/leds/3`. In this case, we use the default options for the subscription because we will work with the default QoS level of 0.

Then, the code specifies the callback function assigned to the `message` event. This function will be called whenever the client receives a message from the MQTT server. The function receives the following two arguments:

- `topic`: This specifies the topic for the received message
- `message`: This is the buffer for the received message's payload

The function saves the results of calling `message.toString` to the `payloadString` variable. It is necessary to call `toString` for `message` because the received `message` argument is a buffer, and we want to work with the received message's payload as a string.

The code prints a message to the console log indicating that a message has arrived with the destination topic and the payload string included in the `topic` argument and the `payloadString` variable.

Then, the code checks whether `topic` starts with `ledCommandBaseTopic` to make sure that the message's destination topic is the one defined for the commands related to LEDs.

Tip

In this case, the code only subscribes to the topic filters that we defined for publishing messages with the commands related to LEDs. However, we include this code to understand how we should check the `topic` value when we subscribe to multiple topics that have different goals with MQTT.js.

Then, the code retrieves the LED ID from the message's topic and saves it in the `ledNumber` variable. The code removes `ledCommandBaseTopic` from `topic` to retrieve the LED ID. For example, if the value for `topic` is `"home/results/leds/3"` the code replaces `"home/results/leds/"` with an empty string and the resulting string, `"3"`, is the LED ID assigned to `ledNumber`.

The next line calls `JSON.parse` with `payloadString` as an argument to parse the message's payload string and create an object from the received JSON string. The line saves the resulting object in the `payload` variable. In this case, we don't validate the content of `payloadString` to make sure that it is a valid JSON in order to keep the example simple.

Then, the code checks whether `ledNumber` and `payload.color` have values. In case they have values, the function prints a message on the console log indicating that it has changed the LED number indicated by the `ledNumber` variable to the color specified in the `payload.color` attribute. In this case, a real-life

application would interact with another Node.js library to change the color for an LED wired to the board. However, we want to stay focused on MQTT over WebSockets.

Then, the code creates a new object named `resultMessagePayload` with "Color" as the key and `payload.Color` as the value. The next line calls the `JSON.stringify` function with `resultMessagePayload` as an argument to generate the JSON string representation of the object and save it in the `resultMessagePayloadString` variable.

Finally, the code calls the `client.publish` function with `ledResultBaseTopic` concatenated with `ledNumber` and `resultMessagePayloadString` as the topic and payload arguments. This way, the code requests the MQTT over WebSockets client to publish a message to the appropriate topic with the results of the execution of the command. In this case, the results indicate the color that has been set to the LED.

Running the home automation web application

Make sure the Mosquitto server or any other MQTT server you might want to use for this example is running. If you aren't working with Mosquitto, make sure you have enabled support for MQTT over WebSockets. Then, execute the following line to start the Node.js example in any computer or device that you want to use as the MQTT client that will process the commands received to control the three LEDs:

```
node leds_control.js
```

The Node.js script will establish a connection with the MQTT server, subscribe to the topics to which the web application will publish the commands for the three LEDs, and display the following message:

I'm connected to the MQTT server

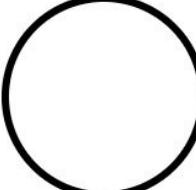
Leave the Node.js script running on the computer or device in which you executed it. Use another computer or device to open a web browser and open the `home_automation.html` web page. The web page will display the following message at the top **Connected with the MQTT Server** after the JavaScript code has established a successful connection with the MQTT Server. The web page will display three textboxes with **FFFFFF** as the initial value for the color and a white background. Whenever we click or tap on these textboxes, we will be able to select a new color for the specific LED. The three circles on the right-hand side of each color picker will be updated to display the color that has been set to the specific LED after the Node.js script processes the received commands based on the selection made in the color picker. The following screenshot shows the initial render for the web page:

MQTT over WebSockets Samp | X

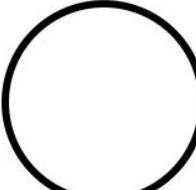
← → ⌂ file:///Users/gaston/Documents/Packt/MQTT/Chapter05/home_automation.html

Connected with the MQTT Server

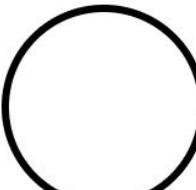
Select color for LED #1



Select color for LED #2



Select color for LED #3



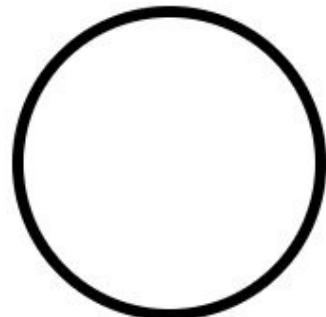
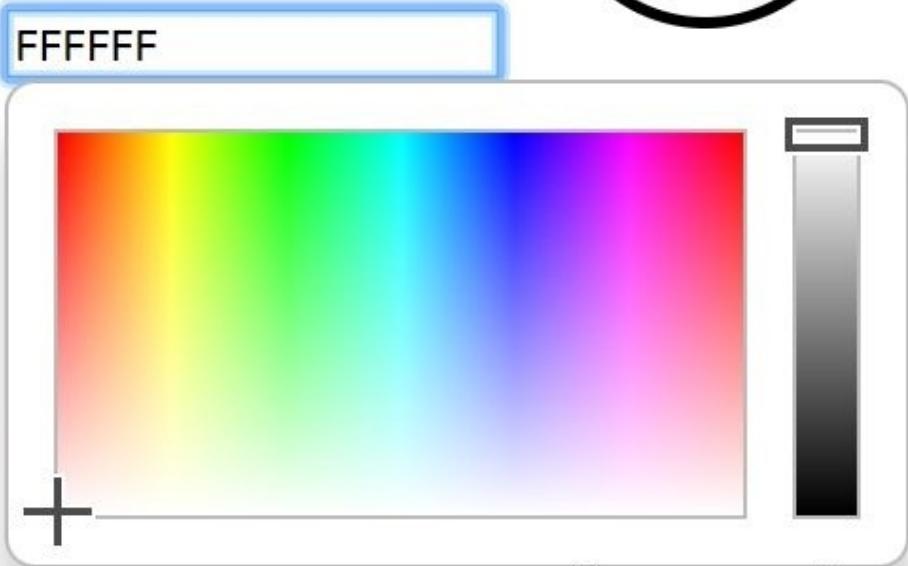
Tip

It is convenient to open the Developer Console for the web browser to see the messages that the JavaScript code will print on the console. This way, you will understand what happens under the hood.

Click or tap on the textbox on the right-hand side of **Select color for LED #1**. The web page will display a color picker, as shown in the following screenshot:

Connected with the MQTT Server

Select color for LED #1 FFFFFF



Select color for LED #2



Select color for LED #3 FFFFFF



Click or tap on a specific color. The background color for the textbox will change to the selected color and the hexadecimal string that represents the RGB color will be displayed as the text.

Go to the window in which you are checking the results of the Node.js script, and you will see messages similar to the following. The value for the `color` key will be different based on your selection in the color picker.

```
Message arrived for topic: home/leds/2, with the following payload:
```

```
{"Color":"ffff00"}
```

```
I've changed LED #2 to ff0000
```

```
Message arrived for topic: home/leds/3, with the following payload:
```

```
{"Color":"00ff00"}
```

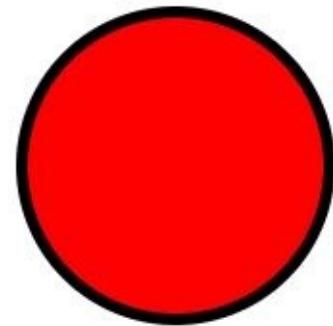
```
I've changed LED #3 to ff0000
```

The Node.js script has received a message in the `home/leds/1` topic with a JSON string as the payload, which requests the color for LED #1 to be set to `ff0000` (255 for red, 0 for green, and 0 for blue). The script processes the command and changes LED #1 to the specified color.

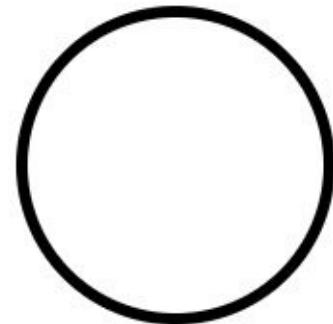
Now, go to the web browser where you left the `home_automation.html` web page. The JavaScript code will receive the message published by the Node.js script to the `home/results/leds/1` topic, and it will update the status message for LED #1 and the related circle's background color. The following screenshot shows the web page that filled the circle for LED #1 with the selected red color:

Connected with the MQTT Server

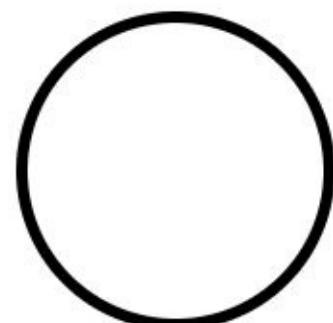
Select color for LED #1 `FF0000`
LED's color set to `#ff0000`



Select color for LED #2 `FFFFFF`



Select color for LED #3 `FFFFFF`



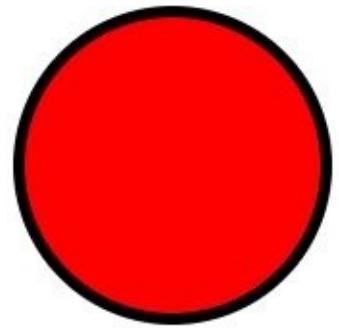
The following screenshot shows the messages displayed in the JavaScript Developer Console for the Chrome browser rendering the web page:

The screenshot shows the Chrome Developer Tools with the 'Console' tab selected. It displays two log entries:

- "I will send ff0000 to LED #1" from [home_automation.js:26](#)
- "Message arrived for topic: home/results/leds/1, with the following payload: {"Color":"ff0000"}" from [home_automation.js:45](#)

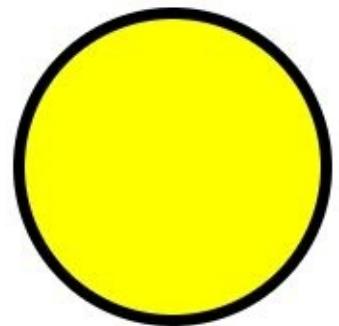
Now, use the color pickers for LED #2 and LED #3 to select two different colors for these LEDs. First, the textbox for the color pickers will change their background colors and then the status messages and the circles will change, as was the case when we selected a different color for LED #1. The following screenshot shows the three LEDs with new colors and their circles updated as a result of the exchange of MQTT messages over WebSockets. We are able to control many LEDs through a web page:

Connected with the MQTT Server



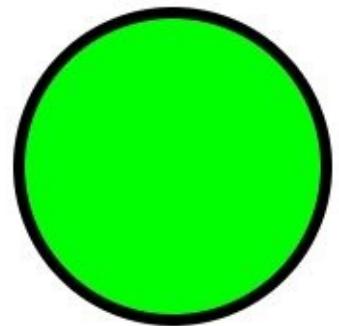
Select color for LED #1 **FF0000**

LED's color set to #ff0000



Select color for LED #2 **FFFF00**

LED's color set to #ffff00



Select color for LED #3 **00FF00**

LED's color set to #00ff00

The following screenshot shows the messages displayed in the JavaScript Developer Console for the Chrome browser rendering the web page:



Elements Console Sources Network Timeline Profiles > ::

 top Preserve log

I will send ff0000 to LED #1	home_automation.js:26
Message arrived for topic: home/results/leds/1, with the following payload: {"Color":"ff0000"}	home_automation.js:45
I will send ffff00 to LED #2	home_automation.js:26
Message arrived for topic: home/results/leds/2, with the following payload: {"Color":"ffff00"}	home_automation.js:45
I will send 00ff00 to LED #3	home_automation.js:26
Message arrived for topic: home/results/leds/3, with the following payload: {"Color":"00ff00"}	home_automation.js:45

>

Go to the window in which you are checking the results of the Node.js script and you will see additional messages similar to the following. The value for the Color key will be different based on your selections in the color pickers.

Go to the window in which you are checking the results of the Node.js script and you will see messages similar to the following. The value for the Color key will be different based on your selection in the color picker:

```
Message arrived for topic: home/leds/1, with the following payload:  
{"Color":"ff0000"}  
I've changed LED #1 to ff0000
```

The following screenshot shows two Terminal windows and Google Chrome running on a computer with macOS. The Terminal on the upper-left corner is displaying an SSH terminal that is running the Node.js script controlling the LEDs on an Intel Joule 570x board. That is, the board is running `leds_control.js`. The window on the lower-left corner is rendering the web page. The Terminal on the right-hand side is displaying the debug messages for a Mosquitto server that is running on the computer:

```

root@intel-corei7-64:~/leds# echo 'Running code on Intel Joule 570x board'
Running code on Intel Joule 570x board
root@intel-corei7-64:~/leds# node leds_control.js
I'm connected to the MQTT server
Message arrived for topic: home/leds/1, with the following payload: {"Color":"ff0000"}
I've changed LED #1 to ff0000
Message arrived for topic: home/leds/2, with the following payload: {"Color":"ffff00"}
I've changed LED #2 to ffff00
Message arrived for topic: home/leds/3, with the following payload: {"Color":"00ff00"}
I've changed LED #3 to 00ff00

```

MQTT over WebSockets Sample

Connected with the MQTT Server

Select color for LED #1 FF0000
LED's color set to #ff0000

Select color for LED #2 FFFF00
LED's color set to #ffff00

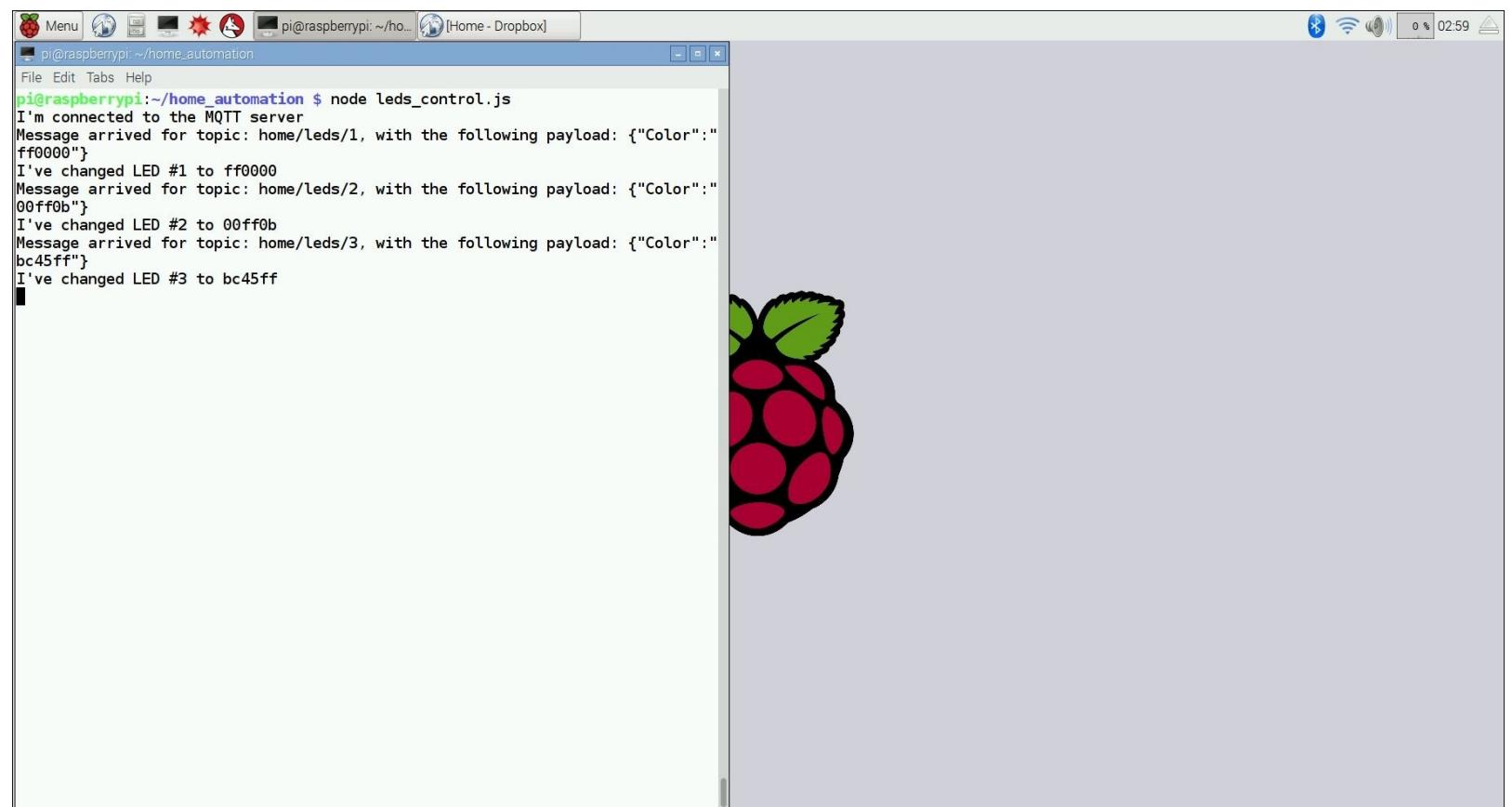
Select color for LED #3 00FF00
LED's color set to #00ff00

```

mosquitto_certs — mosquitto -c /usr/local/etc/mosquitto/mosquitto.conf
/leds/3
1488337888: Sending SUBACK to home-web-82fa2024
1488337901: Received PUBLISH from home-web-82fa2024 (d0, q0, r0, m0, 'home/leds/1', ... (18 bytes))
1488337901: Sending PUBLISH to mqttjs_9451f675 (d0, q0, r0, m0, 'home/leds/1', ... (18 bytes))
1488337901: Received PUBLISH from mqttjs_9451f675 (d0, q0, r0, m0, 'home/results/leds/1', ... (18 bytes))
1488337901: Sending PUBLISH to home-web-82fa2024 (d0, q0, r0, m0, 'home/results/leds/1', ... (18 bytes))
1488337925: Received PUBLISH from home-web-82fa2024 (d0, q0, r0, m0, 'home/leds/2', ... (18 bytes))
1488337925: Sending PUBLISH to mqttjs_9451f675 (d0, q0, r0, m0, 'home/leds/2', ... (18 bytes))
1488337925: Received PUBLISH from mqttjs_9451f675 (d0, q0, r0, m0, 'home/results/leds/2', ... (18 bytes))
1488337925: Sending PUBLISH to home-web-82fa2024 (d0, q0, r0, m0, 'home/results/leds/2', ... (18 bytes))
1488337936: Received PUBLISH from home-web-82fa2024 (d0, q0, r0, m0, 'home/leds/3', ... (18 bytes))
1488337936: Sending PUBLISH to mqttjs_9451f675 (d0, q0, r0, m0, 'home/leds/3', ... (18 bytes))
1488337936: Received PUBLISH from mqttjs_9451f675 (d0, q0, r0, m0, 'home/results/leds/3', ... (18 bytes))
1488337936: Sending PUBLISH to home-web-82fa2024 (d0, q0, r0, m0, 'home/results/leds/3', ... (18 bytes))

```

The next screenshot shows a Terminal window running on the Raspbian operating system on a Raspberry Pi 3 board. The Raspberry Pi 3 is running the Node.js script that controls the LEDs, that is, `leds_control.js`:



Debugging asynchronous code with Chrome Developer Tools (CDT) in JavaScript

Any modern web browser allows us to easily debug JavaScript code. Now, we will use **Chrome Developer Tools (CDT)** to debug the asynchronous JavaScript code that runs in our web application. You can perform the same tasks with other browsers; you will just need to follow different steps.

Make sure that you are using Chrome to render the `home_automation.html` web page. In Chrome, select the Chrome menu located on the upper-right corner of the window and then select **More Tools | Developer Tools**. Chrome will display a window or a panel with the developer tools. The tools can be undocked into a separate window, docked to the bottom of the main window, or docked to the right of the main window.

Click on the **Sources** tab, navigate to `home_automation.js`, and click on this JavaScript file. The source code for the JavaScript file will be displayed.

You can easily establish breakpoints by clicking on the line numbers. Click on the line number for the first line of code within the `onMessageArrived` function. Specifically, click on the following line of code:

```
console.log("Message arrived for topic: " + message.destinationName      + ", with  
the following payload: " + message.payloadString);
```

Now, go to the web browser window where you left the `home_automation.html` web page. Use the color picker for LED #1 to select a different color for this LED. After you select a new color, the Node.js script will send a message with the color that has been set to the LED, the `onMessageArrived` function will be called, and the debugger will hit the breakpoint we have set. The following screenshot shows the debugger hitting the breakpoint:

```

44  onMessageArrived: function(message) { message = Message {}
45  console.log("Message arrived for topic: " + message.destinationName + ", with the following payload: "
46  if (!message.destinationName.startsWith(APP.HomeAutomation.Manager.ledResultBaseTopic)) {
47    return;
48  }
49  var ledNumber = message.destinationName.replace(APP.HomeAutomation.Manager.ledResultBaseTopic, "");
50  var payload = JSON.parse(message.payloadString);
51  if (ledNumber && payload.Color) {
52    // Update the status for the LED that has successfully changed its color
53    var statusLedDiv = document.getElementById("status-message-led-" + ledNumber);
54    //statusLedDiv.style.backgroundColor = "#" + payload.Color;
55    statusLedDiv.textContent = "LED's color set to #" + payload.Color;
56    var statusLedCircle = document.getElementById("status-circle-led-" + ledNumber);
57    statusLedCircle.style.fill = "#" + payload.Color;
58  }
59 },
60
61 onMessageDelivered: function(message) {
62 },
63

```

{ } Line 47, Column 20

You can use the Scope panel on the right-hand side of the debugger to check the values for the different local variables. For example, you can inspect the `message` argument. You can press `F10` to step over the next function call and check the messages printed with `console.log` calls on the **Console** tab. Press `F8` to resume script execution until any other breakpoint is hit.

The usage of the debugger makes it easier to understand when the callbacks are executed and takes advantage of the interactive **Console** to dive deep into all the fields and properties of the different objects that the Paho JavaScript API provides.

Securing MQTT over WebSockets with TLS

So far, we have been working with MQTT over WebSockets without TLS. We can change the configuration and make simple edits to the code to work with MQTT over WebSockets with TLS and enable encrypted communications.

In [Chapter 2, Securing an MQTT Mosquitto Server](#), we secured our Mosquitto server, and therefore, we can use the digital certificates we have created. However, it is very important to know that self-signed certificates require more steps to be installed as trusted certificates in the different operating systems that run the diverse web browsers that can use MQTT over WebSockets.

In this chapter, we saved the following files in a directory called `certificates`. We will use these file names as a baseline for a sample configuration.

- `ca.crt`: Certificate authority certificate file
- `device001.crt`: Client certificate file
- `device001.key`: Client key

In order to work with WebSockets over TLS and enable encrypted communications, it is necessary to follow the next steps:

1. Configure the MQTT server to use WebSockets over TLS. In Mosquitto, it is necessary to specify the values for `cafile`, `certfile`, and `keyfile` settings in the Mosquitto configuration file after the definition of the listener with the WebSockets protocol. Then, it will be necessary to restart the MQTT server.
2. Add the certificate authority certificate file (`ca.crt`) as a trusted certificate to the operating system or web browser. In macOS, it is necessary to add the certificate authority certificate file to Keychain Access to be able to use it. In addition, it is necessary to follow all the steps to make the web browser trust the certificate authority certificate file when we work with self-signed certificates. It is necessary to follow the required procedure for any computer or device that will establish a connection with the MQTT server, including the IoT boards.
3. Set the value for the `useSSL` attribute to `true` in the connection options object we pass as an argument to the `connect` function in the Paho JavaScript client. In the previous code example, you must replace `useSSL: false` with `useSSL: true` in the code that declares the `mqttConnectOptions` object.
4. When you use the Eclipse Paho client JavaScript utility, you must specify `/wss` in **Path** instead of `/ws`. In addition, you must make sure the **TLS** checkbox is activated.
5. Replace `ws` with `wss` in the string that you pass as an argument for `MQTT.js` in `Node.js`. In the previous example, you must replace `var client = mqtt.connect("ws://" + host + ":" + port)` with `var client = mqtt.connect("wss://" + host + ":" + port)`.

In macOS, Linux, or Windows, we add the following lines at the end of the `mosquitto.conf` configuration file.

```
listener 9001 0.0.0.0
```

protocol websockets

In order to make MQTT over WebSockets use TLS, we must add many lines after the previous lines. It is very important that the lines are placed after the previous lines. In [Chapter 2, Securing an MQTT Mosquitto Server](#), you learned that it is a good practice to make a backup copy of the existing `mosquitto.conf` configuration file before making changes to it. This way, we can easily roll back to the previous configuration if something goes wrong.

In macOS or Linux, we must add the following lines at the end of the configuration file and make sure that we replace `/usr/local/etc/mosquitto/certificates` with the full path to the certificates directory we created within the Mosquitto installation folder:

```
cafile /usr/local/etc/mosquitto/certificates/ca.crt  
certfile /usr/local/etc/mosquitto/certificates/server.crt  
keyfile /usr/local/etc/mosquitto/certificates/server.key
```

In Windows, add the following lines at the end of the configuration file and make sure that you replace `C:\Program Files (x86)\mosquitto\certificates` with the full path to the certificates directory we created within the Mosquitto installation folder:

```
cafile C:\Program Files (x86)\mosquitto\certificates\ca.crt  
certfile C:\Program Files (x86)\mosquitto\certificates\server.crt  
keyfile C:\Program Files (x86)\mosquitto\certificates\server.key
```

The `cafile` option specifies the full path for the file that provides a PEM-encoded certificate authority certificate file: `ca.crt`.

The `certfile` option specifies the full path for the file that provides a PEM-encoded server certificate: `server.crt`.

Finally, the `keyfile` option specifies the full path for the file that provides a PEM-encoded server keyfile: `server.key`.

After you make all the changes, the application will be working with MQTT over WebSockets with TLS, and only the web browsers and operating systems that use the certificate authority certificate file will be able to establish a communication with the Mosquitto server.

Test your knowledge

1. The Eclipse Paho JavaScript client provides:
 1. Only a non-blocking (asynchronous) API.
 2. Only a blocking (synchronous) API.
 3. Both a blocking (synchronous) and a non-blocking (asynchronous) API.
2. In the Paho JavaScript API, the `Paho.MQTT.Client` object represents:
 1. The synchronous MQTT over WebSockets client.
 2. The asynchronous MQTT over WebSockets client.
 3. The asynchronous MQTT over TCP client (no WebSockets).
3. After the `Paho.MQTT.Client` instance establishes a connection with the MQTT server, the callback assigned to which of the following attributes of the `connectOptions` object passed as an argument to the `connect` function will be called?
 1. `connectCallback`.
 2. `onSuccess`.
 3. `onConnect`.
4. After the `Paho.MQTT.Client` instance receives a message from one of the topic filters to which it has subscribed, the callback assigned to which of the following attributes of the `connectOptions` object passed as an argument to the `connect` function will be called?
 1. `onMessageReceived`.
 2. `messageReceivedCallback`.
 3. `onMessageArrived`.
5. MQTT.js is an open source library that allows us to work with:
 1. Both MQTT over WebSockets and direct MQTT with Node.js.
 2. Only MQTT over WebSockets with Node.js.
 3. Only direct MQTT with Node.js.

Summary

In this chapter, we analyzed the requirements for controlling LEDs wired to different IoT boards with MQTT messages over WebSockets. We defined the topics that we would use and the commands that would be part of the message's payloads to set the colors for the LEDs.

Then, we created an HTML 5 web page combined with JavaScript and the Eclipse Paho JavaScript asynchronous client to control LEDs. We wrote code to publish messages to target remote devices in JavaScript, and we were able to process the received messages to provide feedback to the user.

We worked with MQTT.js to write a Node.js script to process the commands that simulated the interaction with LEDs on different IoT boards. We were able to run the home automation web application and understand how all the pieces worked together by exchanging MQTT over WebSockets messages. We debugged JavaScript asynchronous code with Chrome Developer Tools to understand the code that the Paho JavaScript client executes when a new message arrives. Finally, we understood the necessary steps to work with MQTT over WebSockets with TLS.

Now that we have understood how to use JavaScript and Node.js to work with MQTT over WebSockets, we will use this protocol to interact with actuators by using the Swift programming language, which is what we are going to discuss in the next chapter.

Chapter 6. Interacting with Actuators in Swift

In this chapter, we will write an iOS app in Swift 3 to control actuators with MQTT messages. In addition, we will use Node.js to write code that runs on IoT boards, receives MQTT messages, and controls motors wired to the boards. We will:

- Understand the requirements to interact with actuators in Swift
- Define the topics and commands to control actuators
- Install the CocoaMQTT client library
- Understand the CocoaMQTT client in Swift 3
- Connect to the MQTT server and subscribe to topics in Swift
- Publish messages to target actuators in Swift
- Process commands that interact with actuators in Node.js
- Run the iOS app that controls an actuator with MQTT
- Run the home automation web application
- Debug events in Swift

Understanding the requirements

In the previous chapters, you learned how MQTT works in detail. We understood how to establish a connection between an MQTT client and an MQTT server. You learned what happens when you subscribe to topic filters and when a publisher sends messages to specific topics. In this chapter, we will use MQTT over TCP again, and we won't use MQTT over WebSockets.

Now, we will use Swift, specifically Swift 3, as our main programming language to generate MQTT clients that will act as publishers and subscribers. We will connect an iOS app as an MQTT client to the MQTT server, and we will process simple commands to control actuators with MQTT messages and display status messages received from the boards that control the actuators.

In this case, we want to create a simple iOS app that we will use at home, and therefore, we won't be working with TLS at all. Then, we will code a Node.js script that will run on the boards that have actuators wired to them. In the previous chapter, we worked with the MQTT.js open source library to use MQTT over WebSockets. However, this library also allows us to work with direct MQTT with Node.js. Hence, we will use this library again to work with MQTT over TCP instead of MQTT over WebSockets. The library is very easy to use and will allow us to process the received messages and publish the results by writing just a few lines of code. The following is the web page for the MQTT.js library:

<https://www.npmjs.com/package/mqtt>. In the previous chapter you learned how to install this library.

We can code to run on many platforms because we will use the same code base to control sensors that use the following IoT boards:

- Raspberry Pi 3
- Raspberry Pi 2 Model B
- Intel Edison
- Intel Galileo Gen 2
- Intel Joule 570x
- Intel Joule 550x

Each platform will work with different kinds of actuators. However, we will focus on the basic features to keep our example simple and to stay focused on MQTT. Then, we will be able to use this project as a baseline for other solutions that require us to run code on an IoT board that runs a Node.js script to use MQTT over TCP, has to connect to an MQTT server, and has to process commands.

The code that runs on boards with actuators wired to them must be able to process commands received in messages in a specific topic. We will use strings with short commands in plain English in the payloads.

We will write a client iOS app in Swift to control one actuator and display the results of turning the actuator on or off. Each IoT board must publish a message to a specific topic whenever a command is successfully executed.

Defining topics and commands

We will use `commands/boards/boardname/actuators/actuatorname` as the topic name to publish the commands for the actuators wired to an IoT board, where `boardname` must be replaced with a unique name assigned to a board and `actuatorname` must be replaced with a unique name assigned to an actuator. For example, if we assign `inteledison01` as the name for an Intel Edison board and `motor01` as the name for a motor controlled with this board, we will have to publish the commands to the `commands/boards/inteledison01/actuators/motor01` topic, and the code that runs on this board will subscribe to this topic to receive messages with commands and react to them. In fact, the board can subscribe to `commands/boards/inteledison01/actuators/+` to receive messages with commands for all the actuators wired to the board named `inteledison01`.

The commands will be sent as strings in plain English. We will have just two valid commands:

- `TURN ON`: Turn on the actuator
- `TURN OFF`: Turn off the actuator

The following line shows an example of the payload for the command that turns on an actuator:

`TURN ON`

We will use `status/boards/boardname` as the topic name to make the boards publish the results of the execution of the received commands, where `boardname` must be replaced with a unique name assigned to a board. For example, if we assign `raspberrypi301` as the name for a Raspberry Pi 3 board, the iOS client app will have to subscribe to the `status/boards/raspberrypi301` topic and the Node.js script that runs on this board will publish, to this topic, the results of the execution of each command.

When an actuator is turned on, the board must publish a message string indicating the actuator name followed by a message that says that the actuator has been turned on. When an actuator is turned off, the board must publish a message string indicating the actuator name followed by a message that says that the actuator has been turned off.

We have all the necessary details to start coding with Swift and then with Node.js. First, we will develop a simple iOS app that can control an actuator, based on the previously explained requirements. Then, we will write a Node.js script that will run on a board and will process the commands generated in the iOS app.

Enabling connections without TLS for a Mosquitto server

Now, we will configure Mosquitto to allow us to establish unsecured connections on port 1883, as it is possible with the default configuration. Take into account that this configuration is not recommended in a production environment and we just make it to work with a simpler configuration in an example that is only going to be executed on our LAN.

In case you are running the Mosquitto server in a Terminal window in macOS or Linux, press *Ctrl+C* to stop it. In Windows, stop the appropriate service. If you are running the Mosquitto server as a service in Linux, run the following command to stop the service:

```
sudo service mosquitto stop
```

Go to the Mosquitto installation directory and open the `mosquitto.conf` configuration file. In [Chapter 2, Securing an MQTT Mosquitto Server](#), you learned that it is good practice to make a backup copy of the existing `mosquitto.conf` configuration file before making changes to it. This way, we can easily roll back to the previous configuration if something goes wrong.

In macOS, Linux, or Windows, add the following lines at the end of the configuration file:

```
listener 1883 protocol mqtt
```

We specified the 1883 value for the `listener` option to make Mosquitto listen for incoming network connections on port 1883. This port is the default one for MQTT without TLS.

The `protocol` option is the next line, after we defined the new listener, and specifies MQTT as the desired protocol to use on port 1883. This way, we enable Mosquitto's support for MQTT over TCP on port 1883 while we still have MQTT over WebSockets on port 9001 and MQTT with TLS on port 8883. Again, we don't want this configuration in a production environment and we are only using it in our LAN.

Save the changes to the `mosquitto.conf` configuration file and launch Mosquitto again with the same mechanism that you learned for Linux, macOS, and Windows in [Chapter 1, Lightweight Messaging with MQTT 3.1.1 and Mosquitto](#). The Mosquitto server will add MQTT without TLS support on port 1883 to the previously configured listeners that will still work with MQTT with TLS on port 8883 and with MQTT over WebSockets on port 9001. It is very important to understand that the certificates we configured for the listener that work on port 8883 and the certificates that work on port 9001 do not apply to the MQTT listener on port 1883. Each listener has its own TLS configuration. In this case, we aren't using TLS for MQTT on port 1883.

Installing the CocoaMQTT client library to create a new iOS app

The CocoaMQTT client library is an open source MQTT version 3.1.1 client library for iOS and macOS, which is written in Swift 3. This client library provides a non-blocking (asynchronous) API. The GitHub page for this client library is <https://github.com/emqtt/CocoaMQTT>.

Tip

You can use the CocoaMQTT client library to develop iOS and macOS apps with Swift 3. You will need a Mac computer and Xcode 8 or greater to run the example.

The easiest way to create a new Xcode project that uses this API is to use the CocoaPods dependency manager for Swift. In case you don't work with CocoaPods, you can easily install it following the detailed instructions included on its main web page, that is, <https://cocoapods.org>.

Once you have CocoaPods installed in macOS, you are ready to create a simple iOS app, based on the Single View Application template with Xcode.

Navigate to **File | New | Project...** in Xcode. Click **iOS** at the top of the **Choose a template for your new project** dialog box. Select **Single View Application** and click on **Next**, as shown in the following screenshot:

Choose a template for your new project:

iOS watchOS tvOS macOS Cross-platform Filter

Application

 Single View Application	 Game	 Master-Detail Application	 Page-Based Application	 Tabbed Application
 Sticker Pack Application	 iMessage Application			

Framework & Library

 Cocoa Touch Framework	 Cocoa Touch Static Library	 Metal Library
--	---	--

Enter ControlActuators in **Product Name**, select **Swift** in language, and select **Universal** in **Devices**, as shown in the next screenshot. This way, we will create an app that can run on iPad, iPod, and iPhone devices. Then, click on **Next**:

Choose options for your new project:

Product Name:

Team: 

Organization Name:

Organization Identifier:

Bundle Identifier: packtpub.com.ControlActuators

Language: 

Devices: 

Use Core Data

Include Unit Tests

Include UI Tests

Cancel

Previous

Next

Select the desired folder in which you want to create the new project folder, make sure that **Don't add to any project or workspace** is selected in the **Add to** drop-down list if this option is shown in the dialog box, and click on **Create**. Xcode will create the new project and all the related files. The following screenshot shows the project navigator located on the left-hand side of the Xcode window:



ControlActuators

ControlActuators

AppDelegate.swift

ViewController.swift

Main.storyboard

Assets.xcassets

LaunchScreen.storyboard

Info.plist

ControlActuatorsTests

Products

Now, let's take a look at the initial code for the two Swift source files included in the `ControlActuators` module:

- `AppDelegate.swift`: This declares the `AppDelegate` class, and it is the entry point to our application
- `ViewController.swift`: This declares the `viewController` class

The following lines show the initial code for the `AppDelegate.swift` source file that declares the `AppDelegate` class without the comments that the template includes in each method:

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        return true
    }
}
```

```
}

func applicationWillResignActive(_ application: UIApplication) {
}

func applicationDidEnterBackground(_ application: UIApplication) {
}

func applicationWillEnterForeground(_ application: UIApplication) {
}

func applicationDidBecomeActive(_ application: UIApplication) {
}

func applicationWillTerminate(_ application: UIApplication) {
}

}
```

The `@UIApplicationMain` attribute included at the top of the declaration of the `AppDelegate` class indicates that the class is designated as the delegate of the shared `UIApplication` object in any iOS app. The `AppDelegate` class is a subclass of the `UIResponder` class and conforms to the `UIApplicationDelegate` protocol. The class declares a `window` stored property of the `UIWindow` type optional (`UIWindow?`) and six instance methods. All the methods receive an `application` argument of the `UIApplication` type, which is another subclass of `UIResponder`. The `application` argument will always be the same instance of `UIApplication` that represents the current iOS app, that is, our app. The `application` method receives a second argument named `launchOptions`, which provides a dictionary with keys indicating the reason that your app was launched for. This method is the only one that has code and just returns `true`.

Tip

By default, Swift 3 externalizes the first parameter. All the methods that receive an `application` argument of the `UIApplication` type in the previous code suppress the externalization of the argument label for the first parameter by adding an underscore (`_`) followed by a space before the parameter label (`application`) in each method's declaration. This way, the code generates methods that we can call without specifying the argument label for the first parameter, that is, with the default behavior that the previous Swift versions used. The `CocoaMQTT` library follows the same approach and adds an underscore (`_`) followed by a space before the first parameter labels in each method's declaration. It is very important to know this because `CocoaMQTT` is written in Swift 3 and you might think it uses the default behavior for Swift 3.

The following lines show the initial code for the `ViewController.swift` source file that declares the `ViewController` class without the comments that the template includes in each method:

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
```

```
    super.viewDidLoad()
}

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
}
}
```

The `ViewController` class is a subclass of the `UIViewController` class and overrides two parameterless instance methods: `viewDidLoad` and `didReceiveMemoryWarning`. Both methods include a line of code that calls the method with the same name in its superclass.

It is important to take into account that the `UIViewController` class, that is, the superclass for `ViewController` is a subclass of the `UIResponder` class and conforms to the following protocols: `NSCoding`, `UIAppearanceContainer`, `UITraitEnvironment`, `UIContentContainer`, and `UIFocusEnvironment`.

Now, close the project we just created in Xcode, and quit Xcode.

Open a Terminal window and go to the folder in which the `ControlActuators.xcodeproj` file was created. Run the following command to ask CocoaPods to generate a `Podfile` file for the current directory. We will modify this file later to specify the dependencies for our project and finally install them with CocoaPods:

```
pod init
```

After you run the previous command, CocoaPods will create a file named `Podfile`. Use your favorite editor to add the following highlighted lines to this file and save the changes:

```
# Uncomment the next line to define a global platform for your project
platform :ios, '9.0'

target 'ControlActuators' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
  # frameworks
  use_frameworks!

  # Pods for ControlActuators
  pod 'CocoaMQTT', '~>1.0.11'

  target 'ControlActuatorsTests' do
    inherit! :search_paths
    # Pods for testing
  end

end
```

Run the following command to install the project dependencies according to the previously modified `Podfile` file. We specified that our project requires `CocoaMQTT` version 1.0.11 or greater. The following command will install all the required dependencies for `CocoaMQTT` and then it will install this library:

```
pod install
```

After CocoaPods downloads and installs all the necessary dependencies, you will see output similar to the following lines:

```
Analyzing dependencies
Downloading dependencies
Installing CocoaAsyncSocket (7.5.1)
Installing CocoaMQTT (1.0.11)
Installing MSWeakTimer (1.1.0)
Generating Pods project
Integrating client project
[!] Please close any current Xcode sessions and use `ControlActuators.xcworkspace` for this project from now on.
Sending stats
Pod installation complete! There is 1 dependency from the Podfile and 3 total pods installed.
```

After the dependencies are installed, CocoaPods will generate a new workspace for the ControlActuators project named `ControlActuators.xcworkspace`. Always open this workspace instead of the project.

Now, launch Xcode and open the `ControlActuators.xcworkspace` workspace. The following screenshot shows the project navigator located on the left-hand side of the Xcode window. Notice that the workspace has two projects: `ControlActuators` and `Pods`. The `Pods` project includes the `CocoaMQTT` library and its dependencies. You can easily check the source code for the different files that compose the `CocoaMQTT` library. The following screenshot shows the project navigator located on the left-hand side of the Xcode window with the two projects:



ControlActuators



iPhone 7 Plus



▼ ControlActuators

▼ ControlActuators

AppDelegate.swift

ViewController.swift

Main.storyboard

Assets.xcassets

LaunchScreen.storyboard

Info.plist

► ControlActuatorsTests

► Products

► Pods

► Frameworks

▼ Pods

Podfile

► Frameworks

▼ Pods

► CocoaAsyncSocket

► CocoaMQTT

CocoaMQTT.h

CocoaMQTT.swift

CocoaMQTTFrame.swift

CocoaMQTTMessage.swift

► Support Files

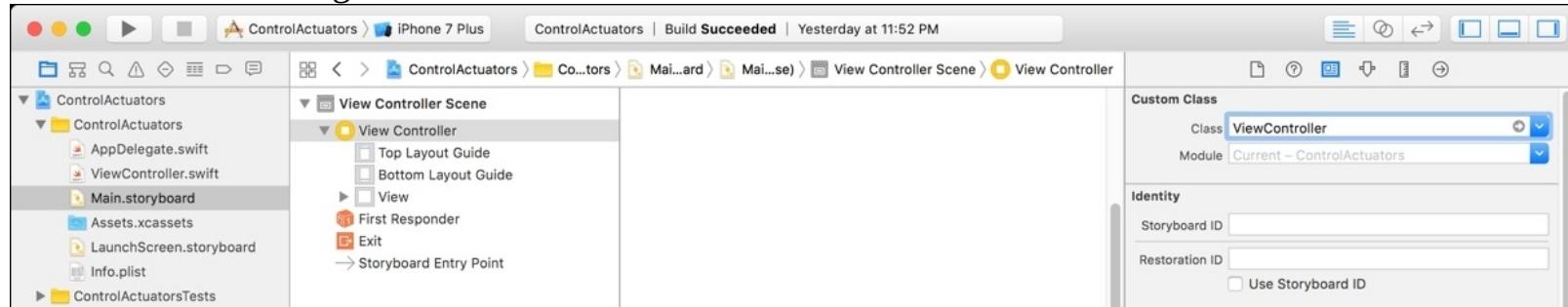
► MSWeakTimer

► Products

► Targets Support Files

Building the UI for the iOS app that will control an actuator

Click on **Main.storyboard** in the Project Navigator on the left-hand side of the Xcode window. The editor will switch to a design view that displays how the view will look. Click on **View Controller** under **View Controller Scene**. Make sure that you see the **Utilities** pane on the right-hand side and check the values for **Identity Inspector**. In order to do so, select **View | Utilities | Show Identity Inspector** or press *Command + Option + 3*. The value for **Class** will be **ViewController** under **Custom Class**, as shown in the following screenshot:



The previously introduced `viewController` class is the custom class associated with the **View Controller** tab in the main storyboard for the iOS app. We will add code to this class later.

Now, we want to add and connect a simple UI element that will allow us to easily turn on and turn off a motor, specifically, a `UISwitch` instance. A switch displays an element that shows the Boolean state of a value. We can tap the control to toggle the value.

Make sure that the **Object Library** tab is visible in **Library View**, which Xcode displays in the bottom half of the **Utilities** pane on the right-hand side. You just need to click on the **Show the Object Library** button at the top of the bottom half. Click on the **Filter** textbox located at the bottom and type `Switch`. **Object Library** will display all the objects that contain `Switch`, and the only one is **Switch**, as shown in the following screenshot:

Document

Label Xcode Specific Label



Object ID UDI-oZ-PCd

Lock Inherited - (Nothing)

Notes A series of icons for note types: three horizontal lines, a dashed line, a solid line, a dashed line with a dot, a red diagonal line, a red diagonal line with a 'T', and a downward arrow.

No Font A font selection bar with a 'T' icon and up/down arrows.

Comment For Localizer



Switch - Displays an element showing the boolean state of a value. Allows tapping the control to toggle the value.

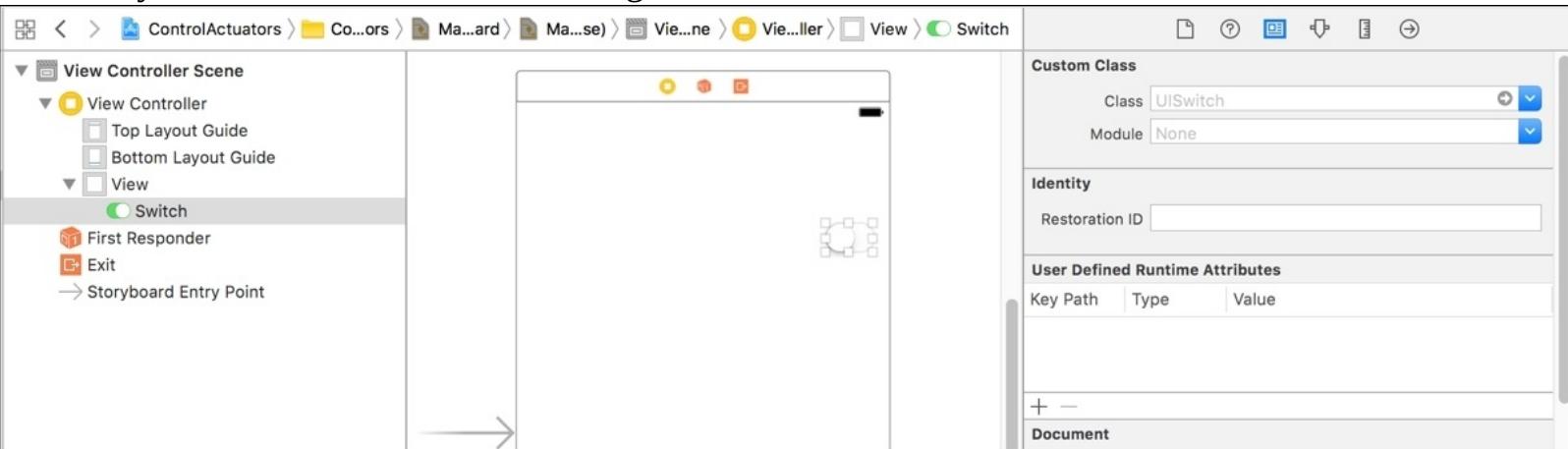


Switch



Drag **Switch** from the previously shown list to the rectangle that defines the view in the preview. Align the new element to the right-hand side. This way, we will have a **Switch** component on the view in the

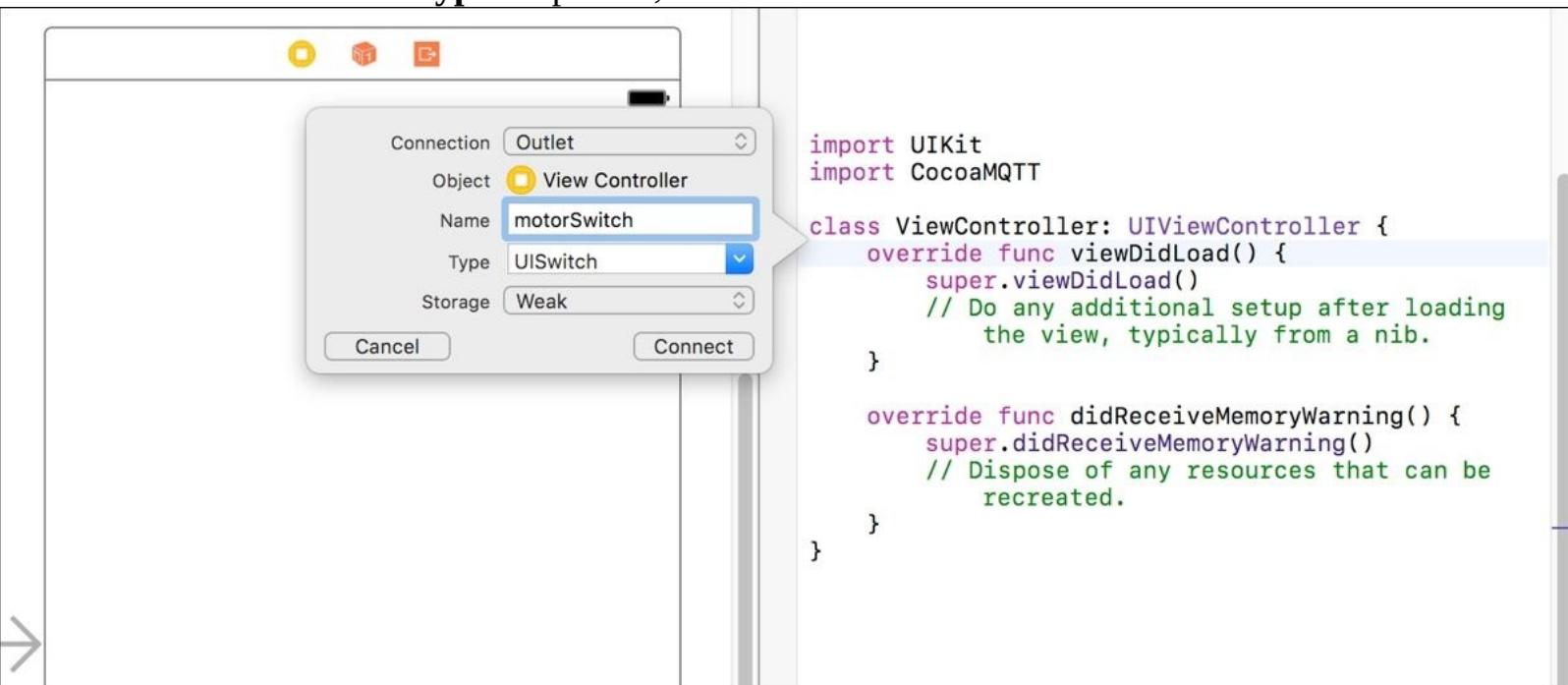
main storyboard, as shown in the following screenshot. Note that the class is **UISwitch**:



We added a **Switch** component on the view. Now, we have to expose the component to make it accessible through code in the previously analyzed **ViewController** class.

Navigate to **View | Assistant Editor | Show Assistant Editor** in the Xcode menu or simply click on the button with two intersecting circles in the upper-right corner (the second button). Xcode will display the source code for the **ViewController** class on the right-hand side of the view preview for the main storyboard.

Press the **Ctrl** key and hold it while you drag the recently added **Switch** component from the view to the blank line after the **ViewController** class declaration. Xcode will display a line and a tooltip with the following legend at the position to which you are dragging the mouse: **Insert Outlet, Action, or Outlet Collection**. Release the **Ctrl** key, and Xcode will display a pop-up dialog box asking us for a name for the new property and **IBOutlet** that it will create. Enter **motorSwitch** in the **Name** textbox, make sure that **UISwitch** is selected in the **Type** dropdown, and then click on **Connect**:



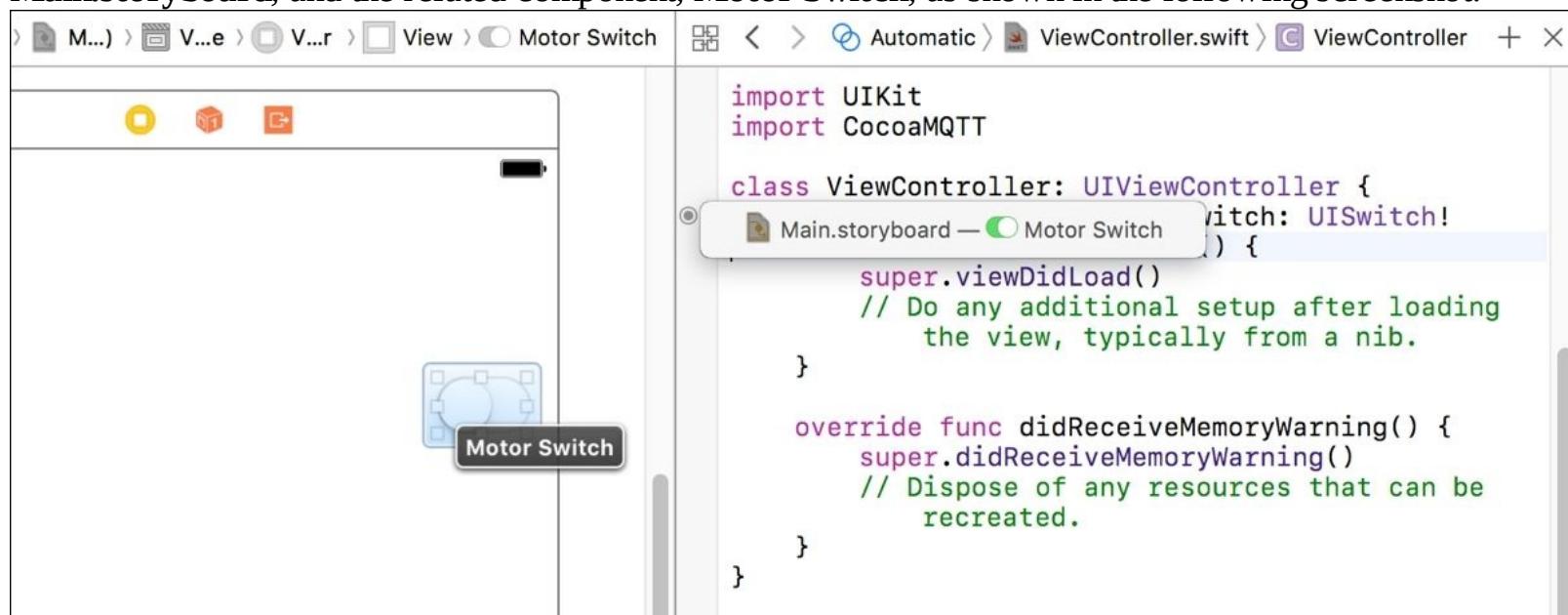
After we click on the **Connect** button, the following highlighted line will appear within the `ViewController` class body. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_06/ControlActuators` folder: class `ViewController`:

```
UIViewController { @IBOutlet weak var motorSwitch: UISwitch!
```

The new line uses the `@IBOutlet` decorator to indicate the outlet connection. The line declares a `motorSwitch` stored property as a weak reference to an implicitly unwrapped optional `UISwitch`. The `weak` keyword indicates Swift to use a weak reference, which allows the possibility of the object that the property points at to become `nil` and avoids retain cycles.

The exclamation mark (!) after the `UISwitch` class name indicates that Xcode wants Swift to treat `motorSwitch` as an implicitly unwrapped optional `UISwitch`. This way, the option will be automatically unwrapped whenever the property is used. However, if it points to `nil`, it will trigger a runtime error.

You will notice there are two small circles on the left-hand side of the new line of code. If you let the cursor hover over this small icon, Xcode will highlight the **Switch** component in the view connected to this property. If you click on the icon, Xcode will display a tooltip with the story board name, `Main.storyboard`, and the related component, **Motor Switch**, as shown in the following screenshot:



We can easily interact with the **Switch** component through the recently added `motorSwitch` property in our `ViewController` class.

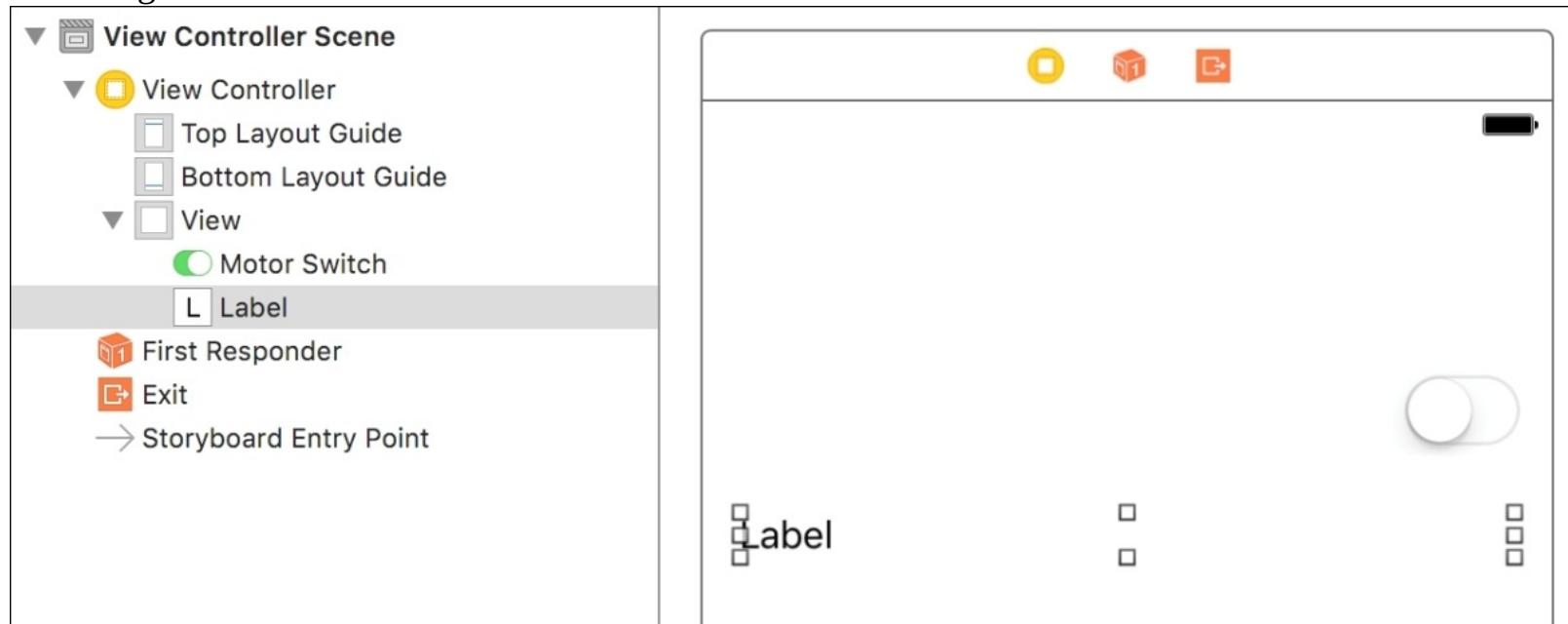
Now, we want to add and connect another simple UI element that will allow us to easily display the status message we will receive from the board after we turn on or turn off the motor with the previously added switch. We will add a `Label` instance. A label displays a variably sized amount of static text. Of course, we are creating a very simple UI to stay focused on MQTT. You will be able to use this example as a baseline and generate a better user experience, combining a great UI with MQTT messages.

Click on `Main.storyboard` in the Project Navigator on the left-hand side of the Xcode window. The editor will switch to a design view that displays how the view will look. Click on **View Controller** under

View Controller Scene. Make sure that you see the **Utilities** pane on the right-hand side and check the values for **Identity Inspector**.

Make sure that the **Object Library** tab is visible in **Library View**. You just need to click on the **Show the Object Library** button at the top of the bottom half. Click on the **Filter** textbox located at the bottom and type **Label**. **Object Library** will display all the objects that contain **Label**, and the only one is **Label**.

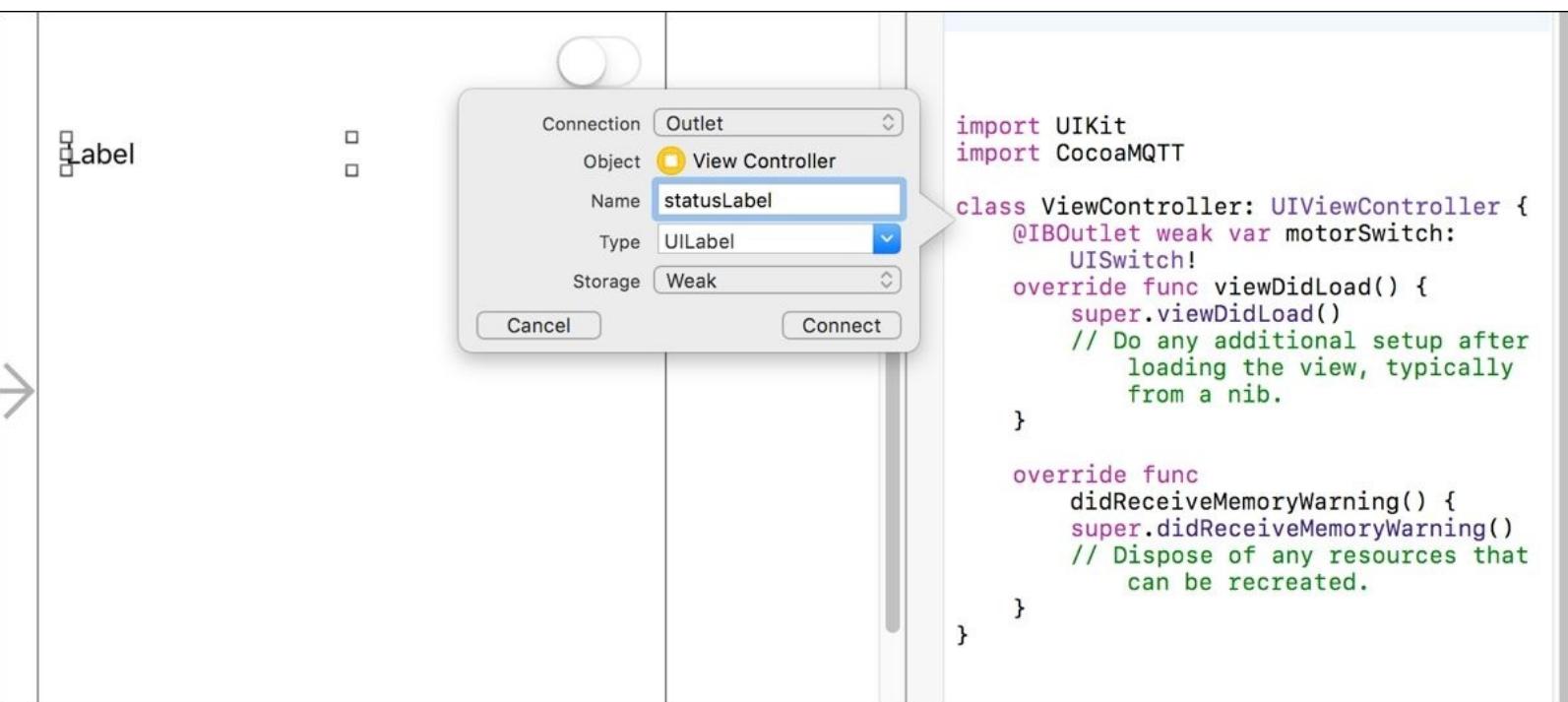
Drag **Label** from the list to the rectangle that defines the view in the preview. Align the new element to the left-hand side and below the previously added switch. This way, we will have a **Label** component on the view in the main storyboard. Note that the class is **UILabel**. Extend the width for the label to reach the right-hand side of the view, as shown in the next screenshot. We want to make sure that the label is capable of displaying the messages we receive from the boards that process the commands, without truncating the text:



We added a **Label** component on the view. Now, we have to expose the component to make it accessible through code in the previously analyzed **ViewController** class.

Navigate to **View | Assistant Editor | Show Assistant Editor** in the Xcode menu or simply click on the button with two intersecting circles in the upper-right corner (the second button). Xcode will display the source code for the **ViewController** class on the right-hand side of the view preview for the main storyboard.

Press the **Ctrl** key and hold it while you drag the recently added **Label** component from the view to the blank line after the **ViewController** class declaration. Xcode will display a line and a tooltip with the following legend at the position to which you are dragging the mouse: **Insert Outlet, Action, or Outlet Collection**. Release the **Ctrl** key, and Xcode will display a pop-up dialog box asking us for a name for the new property and **IBOutlet** that it will create. Enter **statusLabel** in the **Name** textbox, make sure **UILabel** is selected in the **Type** dropdown, and then click on **Connect**:



After we click on the **Connect** button, the following highlighted line will appear within the `ViewController` class body. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_06/ControlActuators` folder:

```
class ViewController: UIViewController { @IBOutlet weak var statusLabel: UILabel! @IBOutlet weak var motorSwitch: UISwitch!
```

The new line uses the `@IBOutlet` decorator to indicate the outlet connection. The line declares a `statusLabel` stored property as a weak reference to an implicitly unwrapped optional `UILabel`.

We can easily interact with the **Label** component through the recently added `statusLabel` property in our `ViewController` class.

Understanding the CocoaMQTT client in Swift 3

The CocoaMQTT client is very easy to use. First, it is necessary to import `CocoaMQTT`, create an instance of the `CocoaMQTT` class, specify the MQTT connection configuration values, and indicate an instance of a class that conforms to the `CocoaMQTTDelegate` protocol. We can call the methods for the created `CocoaMQTT` instance to subscribe to topics and publish messages.

Let's take a look at the code that declares the `CocoaMQTTDelegate` protocol:

```
@objc public protocol CocoaMQTTDelegate { func mqtt(_ mqtt: CocoaMQTT, didConnect host: String, port: Int) func mqtt(_ mqtt: CocoaMQTT, didConnectAck ack: CocoaMQTTConnAck) func mqtt(_ mqtt: CocoaMQTT, didPublishMessage message: CocoaMQTTMessage, id: UInt16) func mqtt(_ mqtt: CocoaMQTT, didPublishAck id: UInt16) func mqtt(_ mqtt: CocoaMQTT, didReceiveMessage message: CocoaMQTTMessage, id: UInt16) func mqtt(_ mqtt: CocoaMQTT, didSubscribeTopic topic: String) func mqtt(_ mqtt: CocoaMQTT, didUnsubscribeTopic topic: String) func mqttDidPing(_ mqtt: CocoaMQTT) func mqttDidReceivePong(_ mqtt: CocoaMQTT) func mqttDidDisconnect(_ mqtt: CocoaMQTT, withError err: Error?) @objc optional func mqtt(_ mqtt: CocoaMQTT, didReceive trust: SecTrust, completionHandler: @escaping (Bool) -> Void) }
```

To make a class conform to the `CocoaMQTTDelegate` protocol, we must code many methods named `mqtt` that receive different arguments. The code we define in these methods with the same name will be executed when certain events related to the MQTT client occur. The usage of the same name might be confusing when we start working, but we can easily understand when each method will be executed by checking the external name of the second argument. The following list explains when each of the methods is executed based on the external name of the second argument. The first argument, named `mqtt`, is the `CocoaMQTT` instance that represents the MQTT client, and we can use it to call the available methods for the client:

- `didConnect`: The `mqtt` method that includes this argument will be executed after the client has established a connection with the MQTT server. The method receives the host and the port for the MQTT server in the `host` and `port` arguments.
- `didConnectAck`: The `mqtt` method that includes this argument will be executed after the connection with the MQTT server has been acknowledged. The method receives the results of the connection acknowledgment in the `ack` argument of the `CocoaMQTTConnAck` enum type. If `ack` is equal to `CocoaMQTTConnAck.accept`, it means that the connection has been successfully accepted by the MQTT server. It is convenient to subscribe to the necessary topic filters by writing the appropriate code in this method. This way, whenever a connection has been accepted, the client will always subscribe to the appropriate topic filters.
- `didPublishMessage`: The `mqtt` method that includes this argument will be executed after a message has been published to a topic. The `message` argument is an instance of the `CocoaMQTTMessage` class. We can use the `message.topic` and `message.string` properties to access the topic and the string representation of the payload for the published message. The `id` argument provides the ID for the message that we can use to track the publish acknowledgement event for the message.
- `didPublishAck`: The `mqtt` method that includes this argument will be executed after the client receives the publish acknowledgement from the MQTT server for the message whose ID is specified

in the `id` argument.

- `didReceiveMessage`: The `mqtt` method that includes this argument will be executed after a message that matches one of the topic filters to which the client has subscribed to has been received. The `message` argument is an instance of the `CocoaMQTTMessage` class. We can use the `message.topic` and `message.string` properties to access the topic and the string representation of the payload for the received message. The `id` argument provides the ID for the message.
- `didSubscribeTopic`: The `mqtt` method that includes this argument will be executed after the client has subscribed to the topic filter specified in the `topic` argument.
- `didUnsubscribeTopic`: The `mqtt` method that includes this argument will be executed after the client has unsubscribed from the topic filter specified in the `topic` argument.

In addition, to make a class conform to the `CocoaMQTTDelegate` protocol, we must code the following three methods:

- `mqttDidPing`: This method is executed after the MQTT client has sent the `PINGREQ` control packet to the MQTT server.
- `mqttDidReceivePong`: This method is executed after the MQTT server has sent the `PINGRESP` response to the MQTT client.
- `mqttDidDisconnect`: This method is executed after the MQTT client disconnects from the MQTT server. The `err` argument is an optional instance that conforms to the `Error` protocol and specifies the details of the error that generated the disconnection.

Connecting to the MQTT server and subscribing to topics in Swift

Now, we will add code to the `ViewController` class to make it conform to the `CocoaMQTTDelegate` protocol. In addition, we will add code to configure the MQTT client and establish a connection with the MQTT server. We will split the code for the `ViewController` class into many code snippets to make it easier to understand each code section. The following lines specify all the necessary imports, start the `ViewController` class declaration, and create many static type properties, the `mqttClient` property, and the `configureMQTTAndConnect` method. The following lines show the code for the `ViewController` class defined in the `viewController.swift` file. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_06/ControlActuators` folder:

```
import UIKit
import CocoaMQTT

class ViewController: UIViewController, CocoaMQTTDelegate {
    @IBOutlet weak var statusLabel: UILabel!
    @IBOutlet weak var motorSwitch: UISwitch!

    static let motor01CommandTopic =
"commands/boards/intel Edison01/actuators/motor01"
    static let motor01StatusTopic = "status/boards/intel Edison01"
    static let turnOnCommand = "TURN ON"
    static let turnOffCommand = "TURN OFF"

    var mqttClient: CocoaMQTT!

    func configureMQTTAndConnect() {
        let clientID = "iOSMotorControl-" + String(ProcessInfo().processIdentifier)
        // Replace with the host name for the MQTT Server
        let host = "localhost"
        // Replace with the port number for MQTT over TCP (without TLS)
        let port = UInt16(1883)
        mqttClient = CocoaMQTT(clientID: clientID, host: host, port: port)
        mqttClient.username = ""
        mqttClient.password = ""
        mqttClient.keepAlive = 60
        mqttClient.delegate = self
        mqttClient.connect()
    }
}
```

The `configureMQTTAndConnect` method generates a client ID for the MQTT client and saves it in the `clientID` constant. The next lines define the `host` and `port` constants and create an instance of the `CocoaMQTT` class named `mqttClient` with `clientID`, `host`, and `port` as the values for the arguments that have the same names. Then, the code sets the values for the `username`, `password`, `keepAlive`, and `delegate` properties. The code specifies `self` as the value for the `mqttClient.delegate` property to specify that we want to use the current instance as the delegate that conforms to the `CocoaMQTTDelegate` protocol. Finally, the code calls the `mqttClient.connect` method to make the MQTT client establish a

connection with the MQTT server.

The following lines show the changes in the `viewDidLoad` method and declare two methods that are required to make the `ViewController` class conform to the `CocoaMQTTDelegate` protocol. The following lines show the code for the `ViewController` class defined in the `ViewController.swift` file. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_06/ControlActuators` folder:

```
override func viewDidLoad() {
    super.viewDidLoad()
    configureMQTTAndConnect()
}

func mqtt(_ mqtt: CocoaMQTT, didConnect host: String, port: Int) {
    // Connection established with the MQTT Server
    print("Connected with MQTT Server \(host):\(\(port))")
}

func mqtt(_ mqtt: CocoaMQTT, didConnectAck ack: CocoaMQTTConnAck) {
    // Connection acknowledged
    print("Connection acknowledged \(ack.rawValue): \(ack.rawValue)")
    if ack == .accept {
        // Subscribe to the motor01StatusTopic topic with QoS 0
        mqtt.subscribe(type(of: self).motor01StatusTopic, qos:
CocoaMQTTQoS.qos0)
    }
}
```

The `viewDidLoad` method calls the previously explained `configureMQTTAndConnect` method after it calls the `super.viewDidLoad` method. This way, after the view is loaded, the app will configure the MQTT client and establish a connection with the MQTT server.

The code in the `mqtt` method with `didConnect` as the external name of the second argument prints a message to the debug output, indicating that the MQTT client is connected with the MQTT server with the host and port received as arguments.

The code in the `mqtt` method with `didConnectAck` as the external name of the second argument prints a message to the debug output, indicating that the connection has been acknowledged with the value received in the `ack` argument. If the value for the `ack` argument is equal to `CocoaMQTTConnAck.accept`, it means that the connection was accepted by the MQTT server and we can subscribe to the desired topic filters. If this happens, the code calls the `mqtt.subscribe` method with the `motor01StatusTopic` type property as the value for the `topic` argument and `CocoaMQTTQoS.qos0` as the value for the `qos` argument. This way, the code requests the MQTT client to subscribe to the desired topic with a QoS level of 0.

The following lines declare the remaining methods that are required to make the `ViewController` class conform to the `CocoaMQTTDelegate` protocol and end the declaration of the class. The following lines show the code for the `ViewController` class defined in the `ViewController.swift` file. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_06/ControlActuators` folder:

```
func mqtt(_ mqtt: CocoaMQTT, didPublishMessage message: CocoaMQTTMessage, id:
```

```

UInt16) {
    print("Message published to topic \(message.topic) with payload \
(message.string!)")
}

func mqtt(_ mqtt: CocoaMQTT, didPublishAck id: UInt16) {
    print("Publish acknowledged with id: \(id)")
}

func mqtt(_ mqtt: CocoaMQTT, didReceiveMessage message: CocoaMQTTMessage, id: UInt16 ) {
    print("Message received in topic \(message.topic) with payload \
(message.string!)")
    if (message.topic == type(of: self).motor01StatusTopic) {
        statusLabel.text = "\(message.string!)"
    }
}

func mqtt(_ mqtt: CocoaMQTT, didSubscribeTopic topic: String) {
    print("Subscribed to \(topic)")
}

func mqtt(_ mqtt: CocoaMQTT, didUnsubscribeTopic topic: String) {
    print("Unsubscribed from \(topic)")
}

func mqttDidPing(_ mqtt: CocoaMQTT) {}

func mqttDidReceivePong(_ mqtt: CocoaMQTT) {}

func mqttDidDisconnect(_ mqtt: CocoaMQTT, withError err: Error?) {
    print("Disconnected from the MQTT Server")
}

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
}
}

```

Most of the methods just print a message to the debug output and some of them are declared with empty code because we don't want to pay attention to those specific events. The code in the `mqtt` method with `didReceiveMessage` as the external name of the second argument prints a message to the debug output indicating that the MQTT client has received a message in a topic with the string representation of the payload. Then, the code checks whether the value of the `message.topic` property matches the value of the `motor01StatusTopic` type property. If these values match, the code sets the `statusLabel.text` property to the string representation of the payload of the received message. This way, whenever the app receives a status update from the board, the app updates the text displayed in the `statusLabel` label.

Publishing messages to target actuators in Swift

Whenever the user changes the value for the `motorSwitch` switch, we want to send a message to request the board to turn on or turn off the motor.

Click on **Main.storyboard** in the Project Navigator on the left-hand side of the Xcode window. The editor will switch to a design view that displays how the view will look. Click on **View Controller** under **View Controller Scene**, then click on **View**, and finally on **Motor Switch**.

Navigate to **View | Assistant Editor | Show Assistant Editor** in the Xcode menu or simply click on the button with two intersecting circles in the upper-right corner (the second button). Xcode will display the source code for the `viewController` class on the right-hand side of the view preview for the main storyboard.

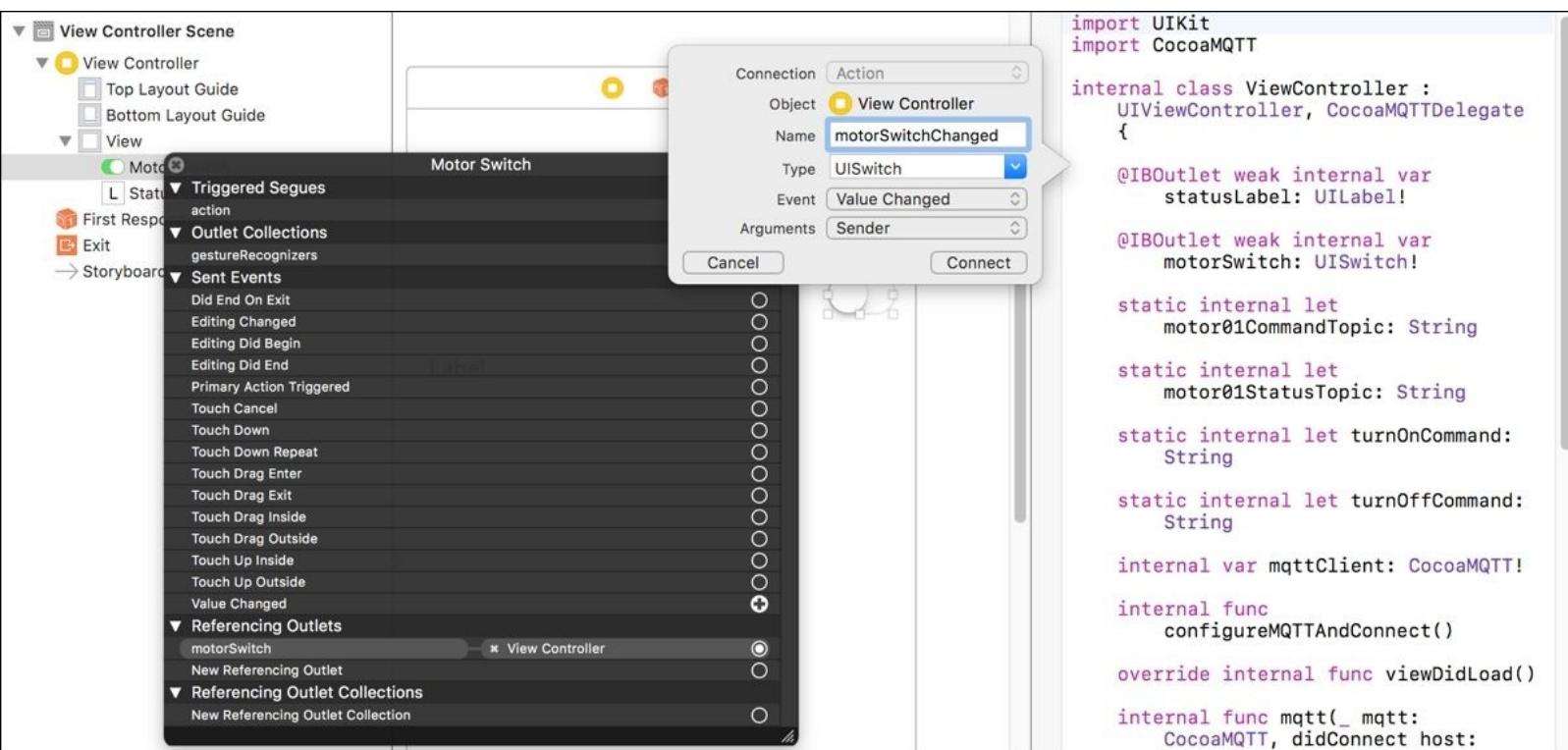
Right-click on **Motor Switch** and Xcode will display a popup that includes the sent events and the associated actions for this component, as shown in the next screenshot:

View Controller Scene

- View Controller
 - Top Layout Guide
 - Bottom Layout Guide
- View
 - Motor Switch
 - Status Label
 - First Responder
 - Exit
 - Storyboard Entry Point



Click and drag the circle on the right-hand side of **Value Changed** from the previously shown popup to the blank line after the `ViewController` class declaration. Xcode will display a line and a tooltip with the following legend at the position to which you are dragging the mouse: **Insert Action**. Release the mouse button, and Xcode will display a pop-up dialog box asking us for a name for the new method and `IBAction` that it will create. Enter `motorSwitchChanged` in the **Name** textbox, make sure that `UISwitch` is selected in the **Type** dropdown, and then click on **Connect**. The following screenshot shows the values specified in the pop-up dialog:



After we click on the **Connect** button, the following highlighted line will appear within the `ViewController` class body. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_06/ControlActuators` folder:

```
class ViewController: UIViewController { @IBAction func motorSwitchChanged(_ sender: UISwitch) {} }
```

The new line uses the `@IBAction` decorator to indicate the action connection. The line declares a `motorSwitchChanged` method that receives the `UISwitch` instance whose value has changed in the `sender` argument.

Now, we will add code to the recently added `motorSwitchChanged` method to send a message to request the board to turn on or turn off the motor, based on the new value that the component has in the `isOn` property. The following lines show the new code for the method. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_06/ControlActuators` folder:

```
@IBAction func motorSwitchChanged(_ sender: UISwitch) { let messagePayload = sender.isOn ? type(of: self).turnOnCommand : type(of: self).turnOffCommand mqttClient.publish(type(of: self).motor01CommandTopic, withString: messagePayload, qos: CocoaMQTTQOS.qos0, retained: false, dup: false) }
```

First, the code declares a `messagePayload` constant that is assigned the value of the `turnOnCommand` type property if the value of `sender.isOn` is `true`. Otherwise, the constant is assigned the value of the `turnOffCommand` type property. Then, the code calls the `mqttClient.publish` method with the `motor01CommandTopic` type property as the value for the `topic` argument, the `messagePayload` constant as the value for the `withString` argument that specified the desired payload for the messages, and

`CocoaMTTQOS.qos0` as the value for the `qos` argument. The other arguments are Boolean values set to `false`. This way, we publish a message with the appropriate command, based on the selected value for the switch. Whenever the user changes the value for the switch, the code in this method will be executed and a message will be sent to the board that controls the motor.

Processing commands that interact with actuators in Node.js

Now that we have coded the iOS app that will allow the user to turn on and turn off a motor, we will write the Node.js code that will control the motor wired to the IoT board. We can run Node.js and the code in any computer or IoT board capable of running Node.js version 6.9.x or higher.

For this example, we specified in the requirements that the motors would be wired to the following IoT boards: Raspberry Pi 3, Raspberry Pi 2 Model B, Intel Edison, Intel Galileo Gen 2, Intel Joule 570x, and Intel Joule 550x. All of these boards can have Node.js installed and run the sample code. However, we can also run the code on many other IoT boards or even on our development computer.

In this example, we will use the MQTT.js open source library to work with direct MQTT with Node.js. Make sure you have Node.js installed in the computer or IoT board you will use to run this example. If you haven't worked with the example in the previous chapter, run the following command to install MQTT.js. Notice that the code is tested with MQTT.js 2.3.1:

```
npm install mqtt
```

Now, we will create a new Node.js JavaScript file named `actuators_control.js`. The code file for the sample is included in the `mqtt_essentials_gaston_hillar_06` folder, in the `actuators_control.js` file:

```
var mqtt = require("mqtt")

// Replace with the host name for the MQTT Server
var host = "localhost"
// Replace with the port number for MQTT over TCP (without TLS)
var port = 1883
var client = mqtt.connect("mqtt://" + host + ":" + port)

var boardName = "inteladison01"
var actuatorCommandBaseTopic = "commands/boards/" + boardName + "/actuators/"
var actuatorResultTopic = "status/boards/" + boardName
var turnOnCommand = "TURN ON"
var turnOffCommand = "TURN OFF"

client.on("connect", function () {
  console.log("I'm connected to the MQTT server")
  client.subscribe(actuatorCommandBaseTopic + "+")
})

client.on('message', function (topic, message) {
  // message is Buffer
  var payloadString = message.toString()
  console.log("Message arrived for topic: " + topic + ", with the following
payload: " + payloadString)
  if (!topic.startsWith(actuatorCommandBaseTopic)) {
    return;
  }
})
```

```

}
var actuatorName = topic.replace(actuatorCommandBaseTopic, "")
if (actuatorName) {
    var resultMessagePayloadString = ""
    if (payloadString === turnOnCommand) {
        console.log("I've turned ON " + actuatorName)
        var resultMessagePayloadString = actuatorName + " has been turned on"
    } else if (payloadString === turnOffCommand) {
        console.log("I've turned OFF " + actuatorName)
        var resultMessagePayloadString = actuatorName + " has been turned off"
    }
    if (resultMessagePayloadString) {
        client.publish(actuatorResultTopic, resultMessagePayloadString)
    }
}
})

```

First, the code requires `mqtt` and saves the object in the `mqtt` variable. Then, the code declares the host and port variables with the host name for the MQTT server and the port number for direct MQTT. Then, the code calls the `mqtt.connect` function with `mqtt://` followed by the host value, a colon (:), and the port number. In case we want to connect to localhost on port 1883, the value for this argument will be `mqtt://localhost:1883`. We specify `mqtt` as the protocol name because we want `MQTT.js` to create a direct MQTT connection. The results of this function are assigned to the `client` variable.

Then, the code specifies the callback function assigned to the `connect` event. This function will be called after the client successfully establishes a connection with the MQTT server.

Tip

After the MQTT client establishes a connection with the MQTT server, `client.connected` will evaluate to `true`.

The function calls the `client.subscribe` function with the result of concatenating "+" to `actuatorCommandBaseTopic` as an argument to subscribe to the `commands/boards/actuators/+` topic filter. In this case, we use the default options for the subscription because we will work with the default QoS level of 0.

Then, the code specifies the callback function assigned to the `message` event. This function will be called whenever the client receives a message from the MQTT server. The function receives the following two arguments:

- `topic`: This specifies the topic for the received message
- `message`: This is the buffer for the received message's payload

The function saves the results of calling `message.toString` to the `payloadString` variable. It is necessary to call `toString` for `message` because the received `message` argument is a buffer and we want to work with the received message's payload as a string.

The code prints a message to the console log, indicating that a message has arrived with the destination topic and the payload string included in the `topic` argument and the `payloadString` variable.

Then, the code checks whether `topic` starts with `actuatorCommandBaseTopic` to make sure that the message's destination topic is the one defined for the commands related to actuators.

Tip

In this case, the code only subscribes to the topic filter we defined for publishing messages with the commands related to actuators. However, we include this code to understand how we should check the `topic` value when we subscribe to multiple topics that have different goals with MQTT.js.

Then, the code retrieves the actuator name from the message's topic and saves it in the `actuatorName` variable. The code removes `actuatorCommandBaseTopic` from `topic` to retrieve the actuator's name. For example, if the value for `topic` is "commands/boards/intel Edison01/motor01", the code replaces "commands/boards/intel Edison01/" with an empty string and the resulting string, "motor01", is the actuator's name assigned to the `actuatorName` variable.

Then, the code checks whether `actuatorName` has a value. In case it has a value, the function checks whether the command included in the message's payload string saved in the `payloadString` variable matches either "TURN ON" or "TURN OFF". If `payloadString` matches any of these values, the code will print a message to the console indicating it has performed the requested operation on the retrieved actuator's name. In addition, the code will save a message to be published in the `resultMessagePayloadString` variable.

Finally, the code calls the `client.publish` function with `actuatorResultTopic` and `resultMessagePayloadString` as the topic and payload arguments. This way, the code requests the MQTT client to publish a message to the appropriate topic with the results of the execution of the command. In this case, the results indicate what happened with a specific actuator.

Running the iOS app that controls an actuator with MQTT

Make sure the Mosquitto server or any other MQTT server you might want to use for this example is running. In case you aren't working with Mosquitto, make sure you have enabled support for direct MQTT without TLS. Then, execute the following line to start the Node.js example in any computer or device that you want to use as the MQTT client that will process the commands received to control the motor:

```
node actuators_control.js
```

The Node.js script will establish a connection with the MQTT server, subscribe to the topics to which the web application will publish the commands for the three LEDs, and display the following message:

```
I'm connected to the MQTT server
```

Leave the Node.js script running on the computer or device in which you executed it. Now, go to Xcode and run the ControlActuators app that we previously coded in the simulator or in a compatible iOS device.

Click or tap the switch to turn the motor on.

Go to the window in which you are checking the results of the Node.js script and you will see messages similar to the following:

```
Message arrived for topic: commands/boards/intel Edison01/actuators/motor01, with  
the following payload: TURN ON  
I've turned ON motor01
```

The Node.js script has received a message in the `commands/boards/intel Edison01/actuators/motor01` topic with a string as a payload that requests the motor named `motor01` to be turned on. The script processes the command and publishes a status message.

Now, go to the simulator or the iOS device in which you are running the iOS app. The MQTT client created by this app will receive the message published by the Node.js script to the `status/boards/intel Edison01` topic and will update the status message with **motor01 has been turned on**. The following screenshot shows the app that changed the status message after the motor was turned on:

Gastons-MacBook-Pro:Chapter06 gaston\$ node actuators_control.js

I'm connected to the MQTT server

Message arrived for topic: commands/boards/inteliedison01/actuators/motor01, with the following payload: TURN ON

I've turned ON motor01

Carrier 7:07 PM

iPhone 7 Plus - iOS 10.0 (14A345)

motor01 has been turned on

You will see the following messages in the Xcode output panel:

Message published to topic commands/boards/inteliedison01/actuators/motor01 with payload TURN ON

Message received in topic status/boards/inteliedison01 with payload motor01 has been turned on

Go to the app and click or tap the switch again to turn the motor off.

Go to the window in which you are checking the results of the Node.js script, and you will see messages similar to the following:

Message arrived for topic: commands/boards/inteliedison01/actuators/motor01, with the following payload: TURN OFF

I've turned OFF motor01

The Node.js script has received a message in the commands/boards/inteliedison01/actuators/motor01 topic with a string as a payload that requests the motor named motor01 to be turned off. The script processes the command and publishes a status message.

Now, go to the simulator or the iOS device in which you are running the iOS app. The MQTT client created by this app will receive the message published by the Node.js script to the status/boards/inteliedison01 topic and will update the status message with **motor01 has been turned off**. The following screenshot shows the app that changed the status message after the motor was turned off:

```
Gastons-MacBook-Pro:Chapter06 gaston$ node actuators_control.js
I'm connected to the MQTT server
Message arrived for topic: commands/boards/intel Edison01/actuators/motor01, with the following payload: TURN ON
I've turned ON motor01
Message arrived for topic: commands/boards/intel Edison01/actuators/motor01, with the following payload: TURN OFF
I've turned OFF motor01
[ ]
```

Carrier 7:12 PM

iPhone 7 Plus - iOS 10.0 (14A345)

motor01 has been turned off

You will see the following messages in the Xcode output panel:

```
Message published to topic commands/boards/intel Edison01/actuators/motor01 with payload TURN OFF
Message received in topic status/boards/intel Edison01 with payload motor01 has been turned off
```

Debugging events in Swift

Now, we will take advantage of Xcode's debugging features to understand how methods are called when events occur.

Leave the application running on the simulator or the iOS device.

Open the source code for `ViewController.swift`.

Click on the left margin of the following line of code to establish a breakpoint at the first line of code in the `mqtt` method with `didReceiveMessage` as the external name of the second argument:

```
print("Message received in topic (message.topic) with payload  
(message.string!)")
```

Now, go to the iOS app and click or tap on the switch to turn on the motor again. After you change the value for the switch, the Node.js script will send a message with the new status for the motor and the `mqtt` method with `didReceiveMessage` as the external name of the second argument will be called. The Xcode debugger will hit the breakpoint we have set. The following screenshot shows the debugger hitting the breakpoint:

ControlActuators > ControlActuators > ViewController.swift > mqtt(_:didReceiveMessage:id:)

```

func mqtt(_ mqtt: CocoaMQTT, didReceiveMessage message: CocoaMQTTMessage, id: UInt16) {
    print("Message received in topic \(message.topic) with payload \(message.string!)")
    if (message.topic == type(of: self).motor01StatusTopic) {
        statusLabel.text = "\(message.string!)"
    }
}

func mqtt(_ mqtt: CocoaMQTT, didSubscribeTopic topic: String) {
    print("Subscribed to \(topic)")
}

func mqtt(_ mqtt: CocoaMQTT, didUnsubscribeTopic topic: String) {
    print("Unsubscribed from \(topic)")
}

func mqttDidPing(_ mqtt: CocoaMQTT) {

}

func mqttDidReceivePong(_ mqtt: CocoaMQTT) {
}

```

Thread 1: breakpoint 1.1

ControlActuators > Thread 1 > 0 ViewController.mqtt(CocoaMQTT, didReceiveMessage : CocoaMQTTMessage, id : UInt16) -> ()

▶ A mqtt = (CocoaMQTT.CocoaMQTT) 0x00006000001b3780
 ▶ A message = (CocoaMQTT.CocoaMQTTMessage) 0x0000600000271200
 ► ObjectiveC.NSObject (NSObject)
 qos = (CocoaMQTT.CocoaMQTTQOS) qos0
 ► dup = (Bool) false
 ► topic = (String) "status/boards/intel Edison01"
 ► payload = ([UInt8]) 26 values
 ► retained = (Bool) false
 A id = (UInt16) 0
 ► A self = (ControlActuators.ViewController) 0x00007fd801d0a520

default_ttl: 1, info_ttl: 0, debug_ttl: 0, generate_symptoms: 0,
 enable_oversize: 1, privacy_setting: 2, enable_private_data: 0
 2017-03-07 19:02:58.060341 ControlActuators[7725:2236090] subsystem:
 com.appleUIKit, category: Gesture, enable_level: 0, persist_level:
 0, default_ttl: 1, info_ttl: 0, debug_ttl: 0, generate_symptoms: 0,
 enable_oversize: 1, privacy_setting: 2, enable_private_data: 0
 2017-03-07 19:02:58.061569 ControlActuators[7725:2236090] subsystem:
 com.appleUIKit, category: GestureEnvironment, enable_level: 0,
 persist_level: 0, default_ttl: 1, info_ttl: 0, debug_ttl: 0,
 generate_symptoms: 0, enable_oversize: 1, privacy_setting: 2,
 enable_private_data: 0
 2017-03-07 19:02:58.062012 ControlActuators[7725:2236090] subsystem:
 com.appleUIKit, category: GestureExclusion, enable_level: 0,
 persist_level: 0, default_ttl: 1, info_ttl: 0, debug_ttl: 0,
 generate_symptoms: 0, enable_oversize: 1, privacy_setting: 2,
 enable_private_data: 0
 Message published to topic commands/boards/intel Edison01/actuators/
 motor01 with payload TURN ON
 Message received in topic status/boards/intel Edison01 with payload
 motor01 has been turned on=Message published to topic commands/
 boards/intel Edison01/actuators/motor01 with payload TURN OFF
 Message received in topic status/boards/intel Edison01 with payload
 motor01 has been turned off
 Message published to topic commands/boards/intel Edison01/actuators/
 motor01 with payload TURN ON
 (lldb)

Auto | ⌂ ⌂ ⌂ Filter All Output ⌂ Filter ⌂ ⌂

You can use all the features included in the debugger to check the values for the different arguments received in the method and the local variables. For example, you can inspect the message argument to understand all the data that the `CocoaMQTTMessage` instance encapsulates.

The usage of the debugger will make it easier to understand when the different methods are executed, and you have a better understanding of the `CocoaMQTT` client. However, it is very important to take into account that the MQTT server will disconnect the client if you pause the code longer than the value specified for the timeout.

Test your knowledge

1. The CocoaMQTT client library is written in:
 1. Objective-C.
 2. Swift 2.
 3. Swift 3.
2. In the CocoaMQTT client library, a `CocoaMQTT` instance represents:
 1. The synchronous MQTT over WebSockets client.
 2. The asynchronous MQTT over WebSockets client.
 3. The asynchronous MQTT over TCP client (no WebSockets).
3. Which of the following methods does the CocoaMQTT client call after a message is received from the MQTT server?
 1. `func mqtt(_ mqtt: CocoaMQTT, didReceiveMessage message: CocoaMQTTMessage, id: UInt16) {`.
 2. `func didReceiveMessage(_ mqtt: CocoaMQTT, message: CocoaMQTTMessage, id: UInt16) {`.
 3. `func mqtt(didReceiveMessage message: CocoaMQTTMessage, id: UInt16) {`.
4. Which of the following properties of a `CocoaMQTTMessage` instance provides the string representation of the message's payload?
 1. `Payload`.
 2. `payloadString`.
 3. `String`.
5. Which of the following properties of a `CocoaMQTTMessage` instance provides the message's destination topic?
 1. `topic`.
 2. `topicString`.
 3. `String`.

Summary

In this chapter, we analyzed the requirements to control actuators wired to different IoT boards with MQTT messages. We defined the topics we would use and the commands that would be a part of the message's payloads to turn the motors on and off.

Then, we created an iOS app with Xcode and Swift 3, combined with the CocoaMQTT client, to control a motor. We wrote code to publish messages to target remote devices in Swift 3, and we were able to process the received messages to provide feedback to the user.

We worked with MQTT.js to write a Node.js script to process commands that simulated the interaction with motors on different IoT boards. We were able to run the iOS app and the Node.js script and understand how all the pieces worked together by exchanging MQTT messages.

We understood how MQTT works in detail and we worked with direct MQTT and MQTT over WebSockets. We worked with unsecured and secured communications. We worked with this protocol in Python, Java, JavaScript, Node.js, and Swift. We were able to create code capable of running in the most popular and powerful IoT boards. We could combine scripts, applications, web applications, and iOS apps with code that controls sensors and actuators on IoT boards. You are now ready to use MQTT in all kinds of projects in the most popular programming languages.

Appendix A. Exersise Answers

Chapter 1, Lightweight Messaging with MQTT

3.1.1 and Mosquitto

Questions	Answers
Q1	2
Q2	1
Q3	3
Q4	2
Q5	3
Q6	1

Chapter 2, Securing an MQTT Mosquitto Server

Questions	Answers
Q1	1
Q2	3
Q3	1
Q4	2
Q5	2

Chapter 3, Controlling a Drone with Python

Questions	Answers
Q1	1
Q2	2
Q3	2
Q4	2
Q5	3

Chapter 4, Interacting with Sensors in Java

Questions	Answers
Q1	3
Q2	2
Q3	1
Q4	3
Q5	3

Chapter 5, Controlling Home Automation in JavaScript

Questions	Answers
Q1	1
Q2	2
Q3	2
Q4	3
Q5	1

Chapter 6, Interacting with Actuators in Swift

Questions	Answers
Q1	3
Q2	3
Q3	1
Q4	3
Q5	1