O'REILLY®

Mobile and Web Messaging

MESSAGING PROTOCOLS FOR WEB AND MOBILE DEVICES

Jeff Mesnil

O'REILLY®

Mobile and Web Messaging

Learn how to use messaging technologies to build responsive and resilient applications for mobile devices and web browsers. With this hands-on guide, you'll use the STOMP and MQTT messaging protocols to write iOS and web applications capable of sending and receiving GPS and device sensor data, text messages, and alerts.

Messaging protocols are not only simple to use, but also conserve network bandwidth, device memory, and batteries. Using this book's step-by-step format, author Jeff Mesnil helps you work with Objective-C and JavaScript libraries, as well as the protocols. All you need to get started are basic programming skills.

- Understand basic messaging concepts and composition
- Learn two common messaging models: point-to-point and publish/subscribe
- Use STOMP to write an iOS application that sends GPS data and a web app that consumes that data
- Build an iOS app with MQTT that tracks and broadcasts device motion data and a web app that displays the data and sends alerts
- Extend STOMP to filter, prioritize, persist, and expire messages
- Take a complete tour of STOMP and MQTT, including features not used in the book's sample apps

"Asynchronous messaging is an essential tool for every software developer. *Mobile and Web Messaging* is an excellent introduction to the world of messaging for mobile and web developers. Through real-world examples, Jeff explains protocols, libraries, and techniques that can help you to connect your mobile and web applications asynchronously."

—Dejan Bosanac
Senior Software Developer at Red Hat

Jeff Mesnil is a software developer who has made significant contributions to many open source middleware projects. He currently holds a Senior Software Engineer position at Red Hat in its JBoss Middleware Division.

MOBILE DESIGN & DEVELOPMENT

US $24.99 CAN $26.99
ISBN: 978-1-491-94480-6

Twitter: @oreillymedia
facebook.com/oreilly

Mobile and Web Messaging

Jeff Mesnil

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Mobile and Web Messaging
by Jeff Mesnil

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Simon St. Laurent and Allyson MacDonald
Production Editor: Nicole Shelby
Copyeditor: Jasmine Kwityn
Proofreader: Gillian McGarvey
Indexer: Ellen Troutman
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

August 2014: First Edition

Revision History for the First Edition
2014-08-08: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491944806* for release details.

978-1-491-94480-6

[LSI]

Table of Contents

A Marion, la lumière qui éclaire mes pas.

Preface

The use of small smart devices is increasing and widening. Smartphones and tablets are connected to the Internet—and automation devices, home devices, car systems, and more are joining them. Even though these devices are increasingly powerful, they have some constraints compared to desktop devices, including limited battery life and memory, as well as intermittent network availability. They require efficient protocols to communicate with other devices and servers while preserving their battery and memory usage.

Messaging technology enables us to overcome some of these constraints by providing protocols that efficiently use the most critical resources (network bandwidth and memory usage) and guarantee that the data will be effectively delivered even in the event of loss of network connectivity.

Applications are increasingly running inside web browsers or using web technology. Messaging concepts apply for these applications too, and can make them more efficient and responsive.

Messaging concepts are well-known in the enterprise software domain, but they are only starting to spread out to other software domains, including mobile and web computing. Developers writing software for these domains need good documentation to leverage these messaging technologies, using them when they make sense while avoiding their pitfalls.

This book provides an introduction to the messaging protocols to help software developers use them in mobile and web applications.

What This Book is About

This is a book about messaging protocols and how software developers can use them to build more responsive, resilient applications running on mobile devices and web browsers.

Messaging protocols are nothing new. They have been used with success in enterprise software for many years, and have been one of the building blocks that let different services and platforms communicate with one another. Their designs make them well-suited to building applications for mobile devices and the Web.

Nowadays, HTTP has emerged as the mainstream transport protocol, and it is extensively used to communicate between clients (from web browsers, of course, but also from desktop and mobile applications, backend services, etc.) and web servers. It has replaced almost all proprietary or nonstandard protocols and is likely to be the de facto choice if your application needs to communicate with any remote endpoint.

Messaging protocols complement HTTP, and this book focuses on cases in which a messaging protocol is better suited than HTTP (or any other request/reply transport protocol) to build mobile or web applications.

What This Book is Not About

Messaging is an overloaded term that can mean many different things in software development. In this book, we will only talk about *application messaging protocols*.

This book is not about *messaging applications* like Apple's Messages or WhatsApp. These applications can be built on top of messaging protocols and this book may be a good introduction if you intend to build one. However, many other features are expected in messaging applications that will not be covered in this book.

This book is also not about *programming language or framework messaging* (as used in Objective-C to invoke methods on an object or in Erlang to communicate between processes).

In the rest of this book, we will use the term *messaging* in the context of application messaging protocol.

Messaging is Simple

At its core, the concept of a messaging protocol is simple:

- An application *produces* a *message* to a *destination* on a *broker*.
- An application subscribes to this same destination to *consume* the message.

In these two sentences, I introduced the five concepts that are (almost) all there is to know about messaging.

A messaging protocol is a simple idea. Most of the complexity of using one is figuring out the best *model* for your applications (how the producers and consumers will exchanges messages). This book will show the two most commonly used models: *point-to-point* and *publish/subscribe*.

Enterprise Messaging Is Not So Simple

When companies are acquired or merged, they need a way to enable communication between their systems. Messaging is one approach to achieve this integration in an unobstrusive way (as much as possible). The systems must agree on the data representation (transmitted in the *message*) and the *destination* (or the topic of interest shared by the different systems).

With its use in enterprise software, messaging protocols became increasingly complex in order to meet enterprise requirements (high availability, failover, load balancing, etc.).

In addition, the integrated applications must often agree on a messaging system to use throughout the company. In the Java world, the specification that deals with messaging is called Java Messaging Service. It defines a set of interfaces that a Java application can use to send and receive messages. However, JMS does not define any protocol (how the bytes are sent over the wire); instead, it leaves this implementation detail to the JMS brokers that implement the API. This means that JMS implementations are not interoperable: one must use the broker's client implementation to send a message to the broker. If applications were using different JMS brokers, they had interoperability issues and must use *bridges* to transfer messages from one JMS broker to another. This lack of interoperability brings complexity, as you need to add servers to host the bridges, make them redundant for high availability, and so on.

Over time, we have seen the appearances of enterprise messaging protocols such as the Advanced Message Queuing Protocol, which handles enterprise features and interoperability. This leads to complex protocols that are difficult to implement and whose interoperability is subjective (backward compatibility is not guaranteed, different implementations may not implement the whole specification leading to interoperability issues).

Mobile Messaging Is Simple Again

The increasing complexity of messaging protocols used in enterprise software makes them difficult to implement, and they are not best suited for constrained environments such as mobile devices.

Mobile devices require messaging protocols with more constrained goals that turn out to be simpler and more efficient to use without draining battery or requiring always-on network connections.

Although these simpler protocols do not provide all the features offered by enterprise-class messaging protocols, they are a good fit for mobile and web platforms.

In this book, I will talk about two of them: STOMP and MQTT.

STOMP really shines for its simplicity if you need to send one text message from any system (operating system, virtual machine, web browser) to another. It is simple enough that a network client such as telnet *is* a STOMP client.

MQTT was created to broadcast data from small devices with low power usage and constrained memory resources. It is well suited for mobile devices, because it preserves battery life and memory.

These protocols are simple to understand and implement. Messaging brokers often provide both of them. For example, a desktop application can use STOMP to send a message and another mobile application can consume the same message using MQTT. The applications are free to use the most appropriate messaging protocol for their needs and rely on the broker implementations for interoperability.

With the advent of mobile devices, we can use these simple messaging protocols to build more reactive, efficient applications. These protocols are available both for mobile and web platforms, so choosing between these two platforms is not constrained by using a messaging protocol.

At the same time, we can leverage these messaging protocols to integrate with legacy systems, too. If the messaging broker also supports an enterprise-class messaging API (such as JMS) or protocols (such as AMQP), we can build mobile and web applications that can consume messages sent from legacy systems.

What's in This Book

In this book, I will introduce messaging protocols for mobile devices and web applications.

Chapter 1, Introduction
: In this chapter, I introduce the concepts of messaging protocols, their models, and message representation. To illustrate the use of messaging protocols on mobile and web platforms, we will build two applications, one using STOMP and the other MQTT. Both applications will come in two parts—one will run on mobile devices, the other inside a web browser; they will communicate using messages. This chapter explains the overall design of these two example applications.

Chapter 2, Mobile Messaging with STOMP
: In this chapter, I present STOMP, a simple text-based messaging protocol. I use StompKit, an iOS library for STOMP, to build the Locations application that sends GPS data from the device and receives text messages.

Chapter 3, Web Messaging with STOMP
: In this chapter, I introduce stomp.js, a JavaScript library for STOMP. I'll present a web application that receives messages from the Locations iOS application and displays the collected GPS data on a map. This web application will also send text messages to the iOS application.

Chapter 4, Advanced STOMP
: In this chapter, I present the advanced features of STOMP that we did not use when building our applications in Chapters 2 and 3. These advanced features are not always used by messaging applications, but they may prove useful as the applications grow in complexity.

Chapter 5, Beyond STOMP
: In this chapter, I discuss features that are not part of STOMP, but are available from some STOMP brokers. These features often help solve common issues and reduce code complexity by leveraging the brokers.

Chapter 6, Mobile Messaging with MQTT
: In this chapter, I introduce MQTT, a binary messaging protocol well suited to broadcast data from mobile or embedded devices. We'll take a look at a mobile iOS application, Motions, that uses MQTT to broadcast data about the device motion using the MQTTKit libary and listen for alerts to change the color of the application.

Chapter 7, Web Messaging with MQTT
: In this chapter, I use MQTT over Web Sockets to write a web application that communicates with the Motions application to display the device motion data and sends alerts to the devices to change their color.

Chapter 8, Advanced MQTT
: In this chapter, I present the advanced features of MQTT that were not used in Chapters 6 and 7.

Appendix A, ActiveMQ
: In this appendix, we explain how to install and configure the messaging broker, Apache ActiveMQ messaging broker, which is used in the book to run the STOMP examples.

Appendix B, Mosquitto

In this appendix, we explain how to install and configure the Mosquitto broker and its command-line tools. Mosquitto is used throughout the book to send and receive MQTT messages.

The book is organized to be read in order, but some chapters can be skipped depending on your experience. Chapter 1 introduces all the concepts discussed throughout the book.

If you are interested in mobile applications, you can focus on Chapters 2 and 6, which present two different messaging protocols for mobile devices. If you are writing web applications, Chapters 3 and 7 are the most relevant.

If you are mainly interested in the STOMP protocol, Chapters 2, 3, 4, and 5 and the most relevant. If you are focused on MQTT, you can read Chapters 6, 7, and 8 instead.

Target Audience

This book is an introduction to the STOMP and MQTT messaging protocols and assumes no prior experience with them; it also explains in detail the messaging protocols. Each platform's clients may provide a different API to deal with the protocols, but the underlying concepts remain the same. For both protocols, we will see two different libraries: an Objective-C library for iOS and a JavaScript library for web applications.

Basic programming skills are required. The examples in the book run on different platforms and I have used the programming language that makes the most sense for each of them.

The mobile applications on iOS will be written in Objective-C. The graphical application requires minimal knowledge of Xcode and Interface Builder, but all the changes are described step-by-step in the book.

The web applications use the JavaScript language. We leverage jQuery to make the web applications interactive and manipulate the page elements, but the messaging code is independent of any JavaScript frameworks.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip, suggestion, or general note.

This element indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. All contents here are licensed under Attribution-NonCommercial-NoDerivatives 4.0 International, and we invite the community at large to contribute work including feature requests, typographical error corrections, and enhancements via our GitHub Issue Tracker. You may reuse any of the text or examples in compliance with the license, which requires attribution. See full license for details.

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/mobile-web-messaging/code/*.

The book contains all the code required to run the examples and the general instructions to set up the user interface of the iOS applications.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Mobile and Web Messaging* by Jeff Mesnil (O'Reilly). Copyright 2014 Jeff Mesnil, 978-1-491-94480-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/mw-messaging*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I'd like to thank the many people who contributed to this book. I hope I have not forgotten anyone, but I probably have.

My colleagues at Red Hat provided help and support in innumerable ways. This list of people is necessarily very incomplete: Bill Burke (I should have listened to you: writing a book is a tiring experience!), Clebert Suconic and Andy Taylor for their messaging expertise, Mark Little for his support. A special thanks to Dimitris Andreadis for his trust and letting me spend part of my work time on this book. Thanks to my teammates in the WildFly and EAP teams (Jason Greene, David M. Lloyd, Brian Stansberry, Kabir Khan, Tomaz Cerar, Emanuel Muckenhuber, and all those I have the pleasure to work with on a daily basis) that helped me lessen my workload so that I could focus on writing this book.

Andy Piper, Clebert Suconic, and Dejan Bosanac helped tremendously by reviewing this book. It is much better thanks to their comments and critics.

The people at O'Reilly were extremely helpful. It was a pleasure working with Allyson McDonald, Nicole Shelby, Rebecca Demarest, Jasmine Kwityn, and Simon St.Laurent.

Thanks to all developers that reported issues on this book and the code examples.

CHAPTER 1

Introduction

In this chapter, I present the main concepts of messaging protocols. To illustrate their use on mobile and web platforms, we will write one application for each messaging protocol described (i.e., one application for STOMP and another for MQTT). This chapter provides an overview of the overall design of the two applications that will be written in the subsequent chapters.

Messaging Concepts

In the Preface, I introduced messaging protocols in two sentences and five concepts:

- An application *produces* a *message* to a *destination* on a *broker*.
- An application subscribes to this same destination to *consume* the message.

Let's now define these five concepts, which are illustrated in Figure 1-1:

Message	The data exchanged between applications
Destination	A type of address that is used to exchange messages
Producer	An application that sends messages *to* a destination
Consumer	An application that consumes messages *from* a destination
Broker	The server entity that will handle messages from producers and deliver them to the consumers according to their destinations

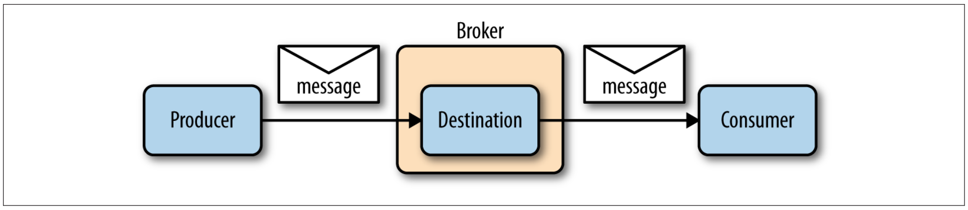


Figure 1-1. Messaging concepts

The simplicity of messaging can be deceiving, but it is this simplicity that allows us to use it in powerful ways.

> Depending on the messaging protocol or model, the *producer* is sometimes called *sender* or *publisher*. Likewise, the *consumer* may be called *receiver* or *subscriber*.
>
> In this book, I will always use the general terms *producer* and *consumer*.

One key aspect of messaging is that it loosely couples its participants. The producer and consumer know nothing about each other. When one application produces a message, it has no knowledge of when or where the message will be consumed. There may be one or many consumers that will receive the message. It is also possible that the message will not be consumed at all if nobody has registered any interest for it.

Likewise, when an application consumes a message, it does not know which application sent it, as they never communicate directly. The consumed message could contain enough information to identify the application, but that is not required (and more often than not, it is not necessary).

Producers and consumers do not even need to be online at the same time. The producers can send a message and exit. The message will be held by the broker until a consumer subscribes to the same destination. At that moment, the broker will deliver the message to the consumer.

Producers and consumers need to know about the broker to connect to it, but they may not even connect to the same broker. A set of brokers can constitute a cluster, and messages can flow from one to another before they are finally delivered to a consumer.

Messaging Models

A messaging model describes how the messages will be routed between the producer and consumers.

There are two main messaging models:

- Point-to-point
- Publish/subscribe

Point-to-Point

In a point-to-point messaging model, a message sent by a producer will be routed to a single consumer.

The producer sends a message to a destination identified as a *queue* in that messaging model. There can be zero, one, or many consumers subscribed (or *bound*) to this queue and the messaging broker will route incoming messages to only one of these consumers to deliver the message. As illustrated in Figure 1-2, when the producer sends a message to the queue, only one of the consumers that are subscribed receives the message.

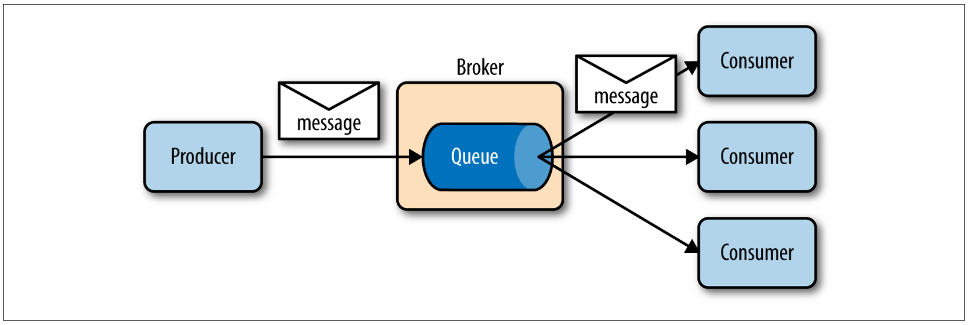


Figure 1-2. Diagram of the point-to-point topology

This is also called a *one-to-one messaging model*: for *one* message sent by a producer to the queue, there is only *one* consumer that will receive it.

If there are no consumers bound to the queue, the broker will retain the incoming messages until a consumer subscribes and then deliver the message to this consumer. Some messaging brokers allow messages to *expire* if they remain in the queue for a certain amount of time. This can be useful to avoid having consumers receive a message corresponding to stale data.

The point-to-point model is best used when only one consumer must process a message. A queue can be used to load balance the message processing across different clients and ensure that only one client will receive it.

Publish/Subscribe

In a publish/subscribe messaging model (often shortened as pub/sub), a message sent by a producer is routed to many consumers.

The producer sends a message to a destination identified as a *topic*. There can be zero or many consumers subscribed to this topic and the messaging broker will route incoming message to *all* these consumers to deliver the message. If there are no consumers bound to the topic, the broker may *not* retain the incoming messages. As illustrated in Figure 1-3, when the producer sends a message to the topic, all the consumers that are subscribed receive the message.

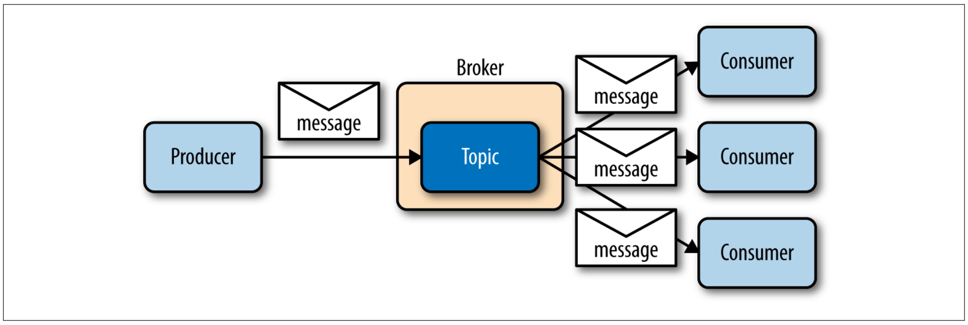


Figure 1-3. Diagram of the publish/subscribe topology

This is also called a *one-to-many* messaging model: for *one* message sent by a producer to a topic, there are *many* consumers that may receive it.

When a message is sent to a topic in this model, we often say that it is *broadcast* to all consumers, as they will all receive it.

Some protocols define the notion of *durable subscribers*. If a consumer subscribes to the topic as a durable subscriber, the broker will retain messages when the consumer is offline and deliver the messages sent to the topic during its downtime when the consumer comes online again.

The publish/subscribe model is particularly suited to send updates. A producer will send a message on the topic with its updated data and any consumer that is subscribed to the topic will be notified of the updates.

Message Representation

Producers and consumers exchange information using messages.

As illustrated in Figure 1-4, a message is composed of three separate pieces of data: destination, headers, and body.

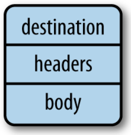


Figure 1-4. Diagram of a message

When a producer sends a message to a *destination*, the name of the destination is put inside the message. When a consumer receives this message, it can use this information to know which destination held this message. This is especially useful when a consumer is subscribed to many destinations, as it helps identify the exchanged data.

A message may also contain *headers*. Messaging protocols use headers to add metadata information to the messages. This metadata can be read by the consumers and gives additional contextual information to a message. Examples of such metadata include message identifiers (that uniquely identify a message for a broker), timestamps (the date and time it was sent by the producer), and redelivered flags (if the message is being delivered again after a failed first attempt). Headers are specific to messaging protocols. Some messaging protocols (such as STOMP) allow the producer to set application-specific headers in addition to the headers defined by the protocol. Other protocols (such as MQTT) do not allow producers to set application-specific headers. In that case, the producer has to put any application-specific information in the message payload.

Finally, a message can have an optional *body* (or payload) that contains the data exchanged between the producer and consumer. The type of body depends on the messaging protocols, some defining text payload (such as STOMP) or binary (MQTT). A payload is an *opaque blob of content*. The broker does not read or modify it when it routes a message.

In most cases, we will only use the message body to pass information using a variety of formats (JSON string, simple plain string, array of float values, etc.). However, if the protocol permits it, we will also set additional headers to the message to give metadata information about the body (its content type, its length, etc.) or activate some broker features.

Examples

To illustrate the use of messaging protocols on mobile and web platforms, we will build two sets of applications in this book. Each set will be be composed of an iOS application and a web one. The first application will use STOMP and the second one will use MQTT.

Locations Application Using STOMP

Suppose that we work for a delivery company that uses a fleet of trucks to deliver packages to its clients.

Each truck is responsible for the delivery of packages and the drivers receive orders from the company's headquarters. To efficiently manage all the trucks, the company wants to monitor the truck's positions and be able to adjust the orders as necessary.

We will build a very simple application named Locations that looks similar to this example (see Figure 1-5).

The "truck" will use an iOS application to broadcast the device's geolocation data using its GPS sensor and display text messages that it receives from the headquarters. The "headquarters" will use a web application to display the location of all the trucks on a map. It will also be used to send text messages to a given truck on their devices:

- In Chapter 2, we will write the Locations iOS application using the STOMP protocol to send GPS data and receive text from an iOS device.
- In Chapter 3, we will write the web application using the STOMP protocol over Web Sockets to receive GPS data in a web browser and send text to the devices.

Before introducing STOMP, the messaging protocol that will be used by the application, we can already define the application's messaging models and how the different parts of the application will exchange messages.

Messaging models for the Locations application

In this application, we will use two destinations, one with a publish/subscribe model to broadcast the device GPS data and one with a point-to-point model for the text messages:

- `device.XXX.location` is the *topic* to broadcast its GPS geolocation data where `XXX` is the device identifier (publish/subscribe model)
- `device.XXX.text` is the *queue* to receive simple text messages (point-to-point model)

A topic is used to send the GPS data, as this allows potentially many consumers to receive the information. However, a queue is used to handle the device's text as only one single device will consume messages from this destination.

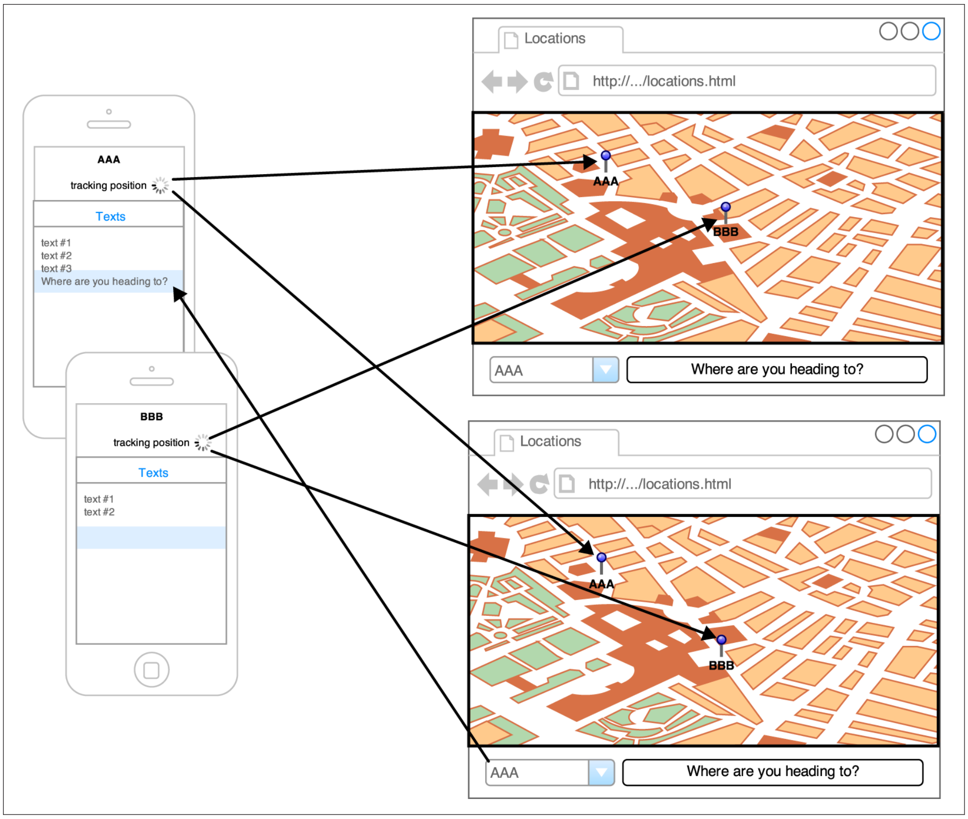

AAA
tracking position
Texts
text #1
text #2
text #3
Where are you heading to?
BBB
tracking position
Texts
text #1
text #2
Locations
http://.../locations.html
AAA
BBB
AAA
Where are you heading to?
Locations
http://.../locations.html
AAA
BBB
AAA
Where are you heading to?

Figure 1-5. Diagram of the Locations application with two devices, AAA and BBB, and two web applications

Each device (identified by a unique identifier `XXX`) running the Locations application will be:

- A *producer* of messages to the topic `device.XXX.location`
- The only *consumer* of messages from the queue `device.XXX.text`

Conversely, the web application will be:

- A *consumer* of messages from *all* the topics of the form `device.XXX.location`
- A *producer* of message to *all* the queues of the form `device.XXX.text`

Message representation for the Locations application

There will be two types of exchanged messages:

- One to represent GPS data (exchanged on the topics `device.XXX.location`)
- One to represent orders (exchanged on the queues `device.XXX.text`)

The Locations iOS application will send the GPS data using a JSON representation in the message payload (Example 1-1).

Example 1-1. Geolocation message payload

```
{
  "deviceID": "BBB", ❶
  "lat": 48.8581, ❷
  "lng": 2.2946, ❸
  "ts": "2013-09-23T08:43Z" ❹
}
```

❶ `deviceID` is the identifier of the device that sends its position

❷ `lat` is a number representing the position's *latitude*

❸ `lng` is a number representing the position's *longitude*

❹ `ts` is a string representing the time when the position was taken (using the ISO 8601 format)

The text messages that will be consumed by the Locations iOS application will be represented as a simple plain-text string, as shown in .

Example 1-2. Text message payload

```
"Where are you heading to?"
```

A more realistic representation of this message would also contain additional information such as the identifier of the headquarters sending the message. We are using a plain-text version, because this format is sufficient for our preliminary example.

With the messaging topology and data representation known, we can now refine the Locations application diagram (Figure 1-6).

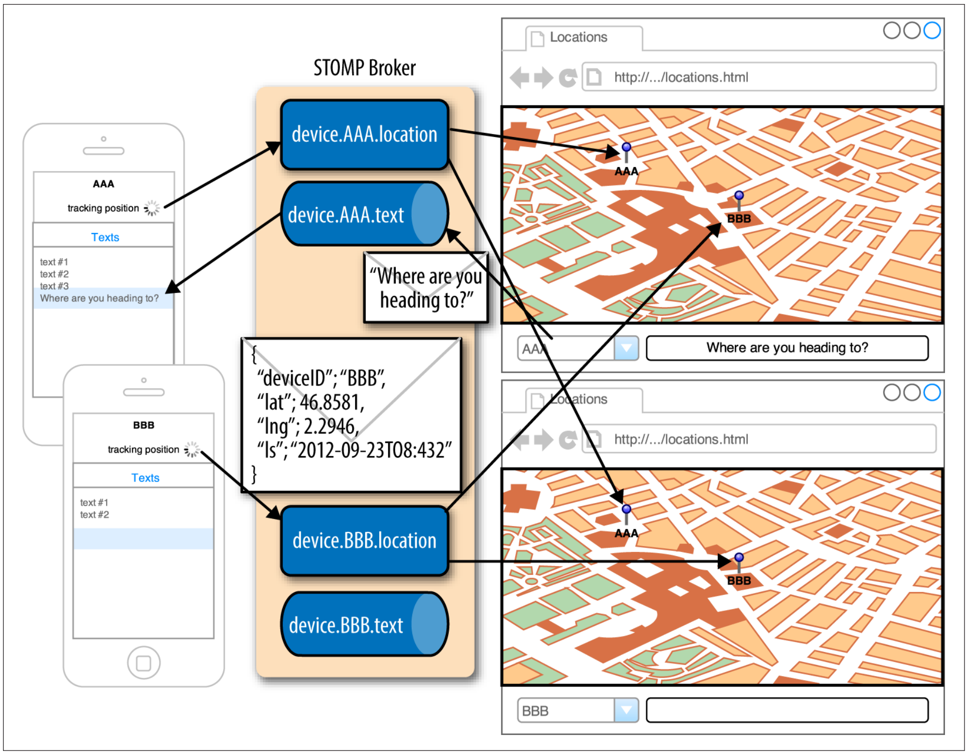


Figure 1-6. Diagram of the Locations application with its messaging models and representations

Motions Application Using MQTT

Most mobile devices contain a sensor that allows for tracking the device motions (and to a certain extent, the motions of the user). Using additional sensors, we could even conceivably monitor the user's health data (heart rate, hydration, blood glucose level, etc.). This type of information could be sent to a centralized application that would be able to track the data and send alerts to the user when necessary.

Following this model, we will write a simple application called Motions to illustrate the use of the MQTT protocol. The iOS application will track the device motion and change its background color to advise its user when an alert message is received.

Device motion will be represented by three values corresponding to its pitch, roll, and yaw values, as shown in Figure 1-7.

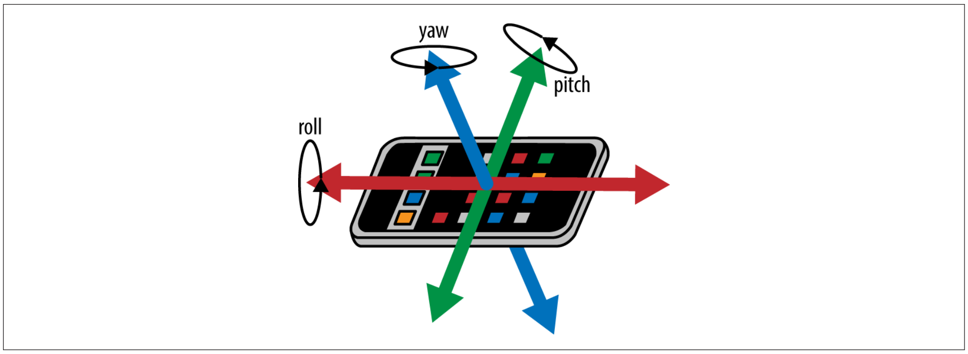


Figure 1-7. The pitch, roll, and yaw values represent the device motion

The pending web application will display the motions of all devices that broadcast them and be able to send alert messages to them (Figure 1-8):

- In Chapter 6, we will write the Motions iOS application using the MQTT protocol to send data about the device motions and receive alerts.
- In Chapter 7, we will write a web application using the MQTT protocol over Web Sockets to receive all the device motions data and display them. The web application will also send alert messages to any devices sending its motions data.

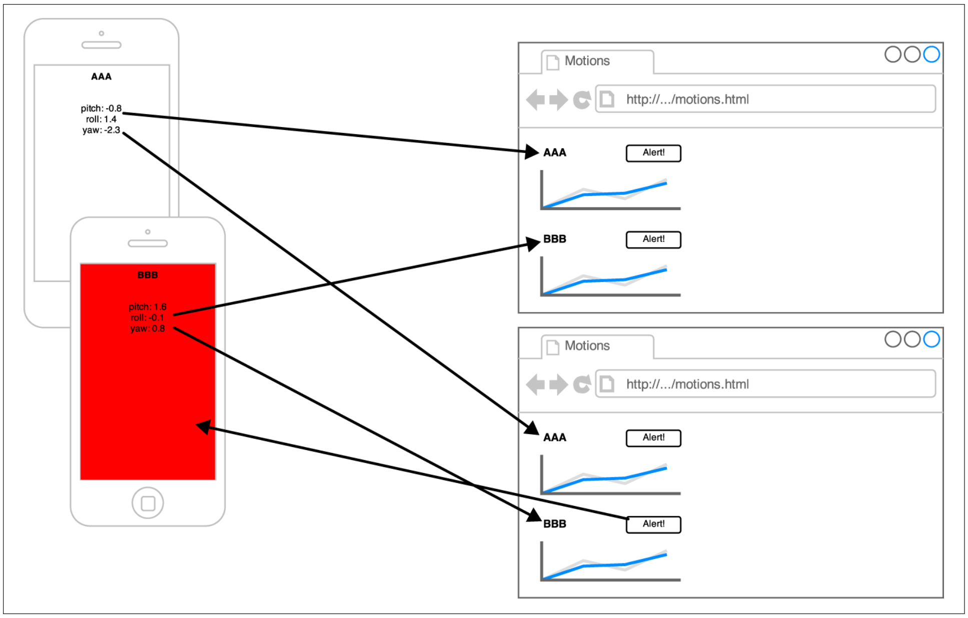


Figure 1-8. The Motions application with two clients, AAA and BBB, and two web applications monitoring them

Messaging models for the Motions application

In this application, we will use two destinations with the publish/subscribe model:

- `/mwm/XXX/motion` (where `XXX` is the device identifier) is the topic to broadcast the device motion data (publish/subscribe model)
- `/mwm/XXX/alert` is the topic to exchange alert messages for a given device (publish/subscribe model)

Each device running the Motions application will be:

- A *producer* of messages to the topic `/mwm/XXX/motion`
- A *consumer* of messages from the topic `/mwm/XXX/alert`

Conversely, the web application will be:

- A *consumer* of messages from all the topics of the form `/mwm/XXX/motion`
- A *producer* of message to all the topics of the form `/mwm/XXX/alert`

MQTT only supports the publish/subscribe messaging model. Ideally, the alert destination would be better modeled as a queue (one per device). MQTT does not have support for queues, so we will work around that by using one topic for each device and only have the corresponding device subscribe to it.

Message representation for the Motions application

There will be two types of exchanged messages:

- One to represent device motion data (exchanged on the topics `/mwm/XXX/motion`)
- One to represent alerts (exchanged on the topics `/mwm/XXX/alert`)

The Motions iOS application will send the device motions data in a binary message where its payload will be composed of three 64-bit floats representing the device's pitch, roll, and yaw values (Example 1-3).

Example 1-3. Device motion message paylaod

```
<< 1.6 -0.1 0.8 >> ❶
```

❶ The message is composed of three 64-bit floats for the pitch, roll, and yaw values.

The alert messages that will be consumed by the Motions iOS application will be represented as a simple plain-text string corresponding to a color. The Motions application will use this payload to temporarily change its background color to alert the user (Example 1-4).

Example 1-4. Alert message payload

```
"red" ❶
```

❶ The message is composed of a string containing the name of a simple color.

With the messaging topology and data representation known, we can now refine the Motions application diagram (Figure 1-9).

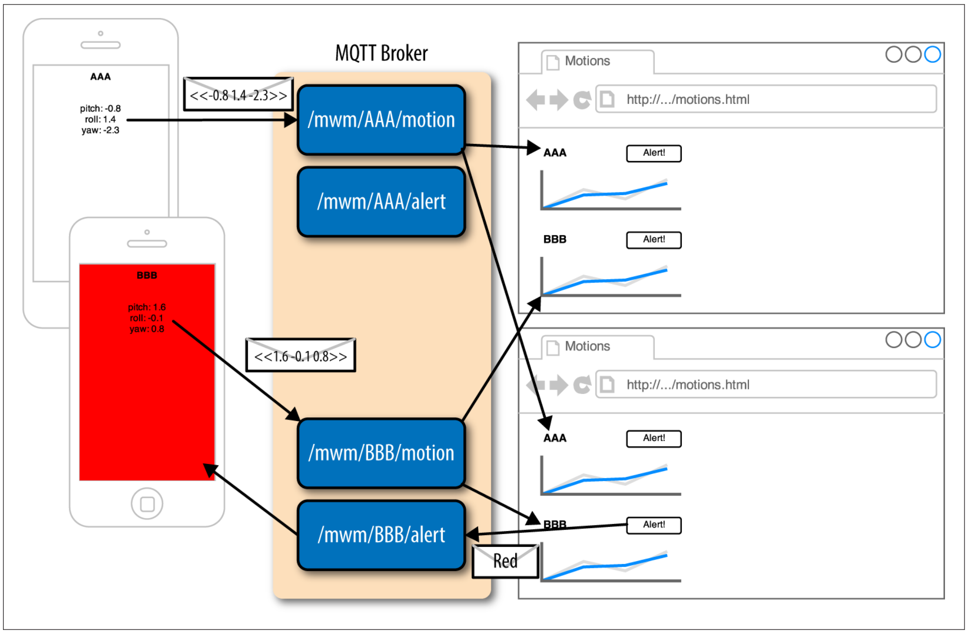


Figure 1-9. Diagram of the Motions application with its messaging models and representations

Summary

In this chapter, we learned about the messaging protocols and how they differ from request/reply protocols. We introduced two messaging models (point-to-point and publish/subscribe) and the parts that compose a message (destination, headers, and payload).

We also took a look at the two applications that will be written in the subsequent chapters—the Locations application (which uses STOMP) and the Motions application (which uses MQTT)—and established the messaging models and representation that they will use.

In the next chapter, we will start writing the Locations iOS application. If you are more interested in learning about MQTT, you can go directly to Chapter 6 to start writing the Motions application.

PART I
STOMP

For the next four chapters, we will use the STOMP protocol to build our applications (this book covers the latest version of the protocol, STOMP 1.2). STOMP is a simple text-based messaging protocol that is well suited to develop lightweight messaging applications on any platform. It provides an interoperable wire format so that any client can communicate with any message broker. The simplicity of the protocol ensures that it is straightforward to have interoperability between client and brokers. It does not define the semantics of the destination; they depend on the STOMP broker you are using.

There are some conventions shared by STOMP brokers (e.g., prefixing a destination by `/queue/` to use a queue and by `/topic/` to use a topic), but you need to consult your broker documentation to check which messaging models are supported (including *point-to-point* and *publish/subscribe*) and how to use them.

STOMP is based on text and a few parsing rules, which makes it simple to use from any platform able to read and write text and open a network connection. However, being text-based means that the protocol is not the most efficient to use, as it requires more network bandwidth and memory than corresponding binary-based protocols. If your applications can work with these constraints, STOMP is a good messaging protocol to use.

Before using a STOMP client, a broker must be installed and configured to be able to exchange messages. In this book, we use Apache ActiveMQ, and Appendix A shows how to install and configure it. When ActiveMQ is started, it will accept STOMP connections on the `61613` port.

CHAPTER 2

Mobile Messaging with STOMP

In this chapter, we will write our first messaging client: a native application running on an iPhone. We will use STOMP to send and receive messages using the Objective-C library StompKit.

In "Locations Application Using STOMP" on page 6, we described the Locations application. In this chapter, we will write the iOS application that broadcasts the device's position and receives text messages (Figure 2-1).

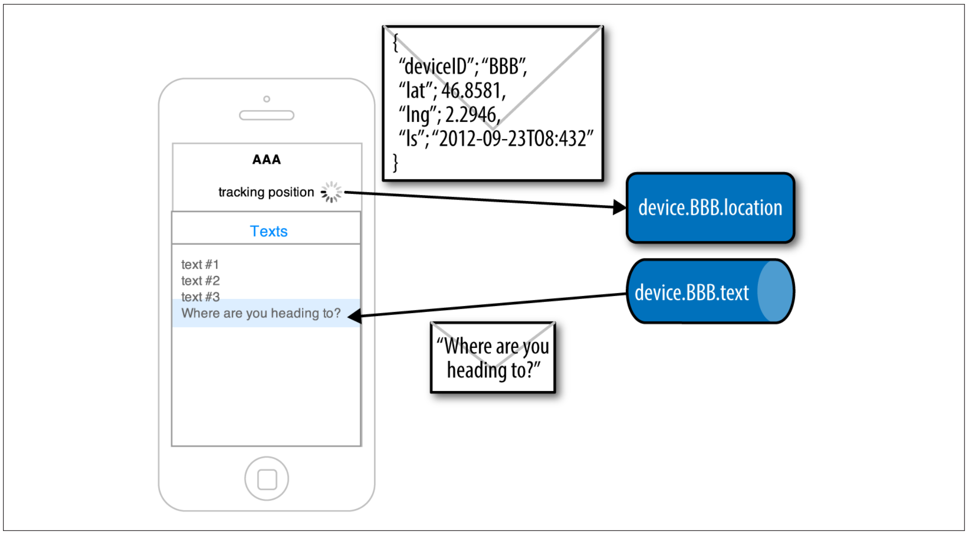


Figure 2-1. Diagram of the Locations iOS application

Throughout the chapter, we will show all the code required to run the application.

The whole application code can be retrieved from the GitHub repository in the *stomp/ios/* directory.

StompKit

To use STOMP on iOS, we will use the StompKit Objective-C library that implements the STOMP protocol in a modern, event-driven way using ARC, Grand Central Dispatch, and blocks.

The source code of this library project is hosted on GitHub.

Create the Locations Project with Xcode

We will use Xcode to create the `Locations` iOS application. When Xcode is installed and started, we create a new project from its launch screen, as illustrated in Figure 2-2:

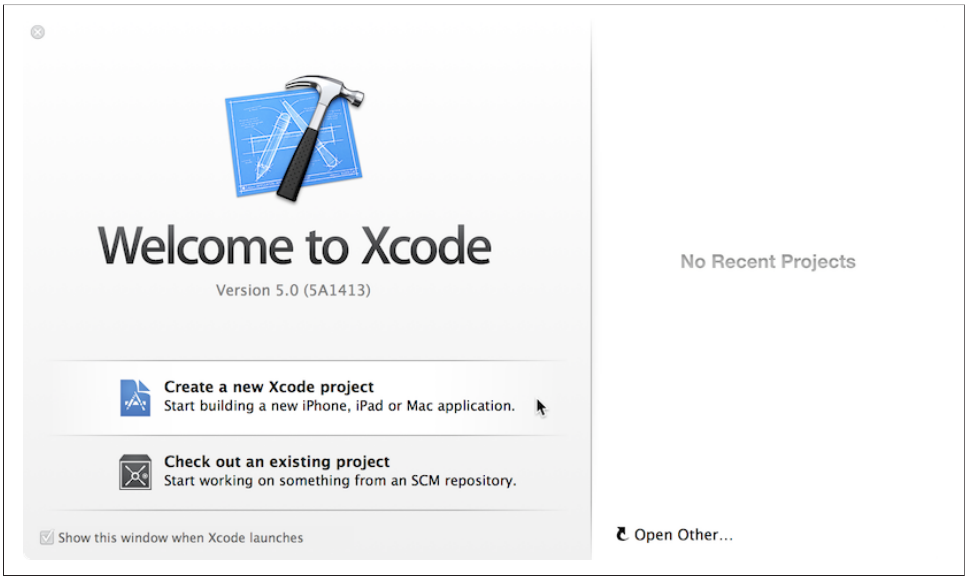
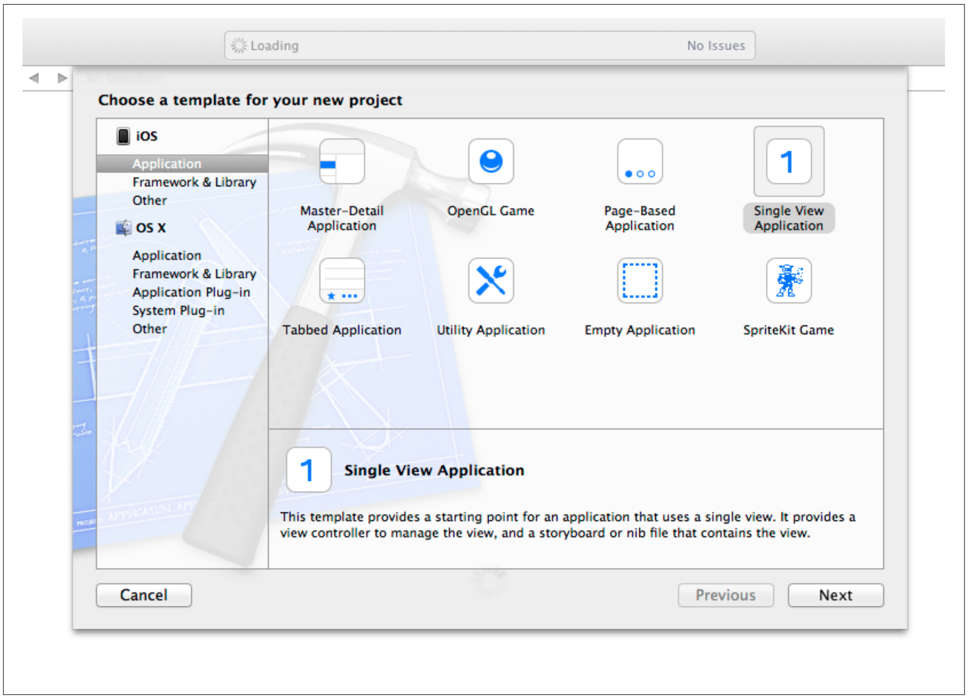

Welcome to Xcode
Version 5.0 (5A1413)
Create a new Xcode project
Start building a new iPhone, iPad or Mac application.
Check out an existing project
Start working on something from an SCM repository.
Show this window when Xcode launches
No Recent Projects
Open Other...

Figure 2-2. Select "Create a new Xcode project" from the Xcode launch screen

The application consists of a single view, so we choose the Single View Application template in iOs → Application from the template screen, as illustrated in Figure 2-3. We will call the project Locations and select to build it only for iPhone devices, as illustrated in Figure 2-4.



This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

Figure 2-3. Select Single View Application from the template screen

Finally, we will save it in a folder on our machine.

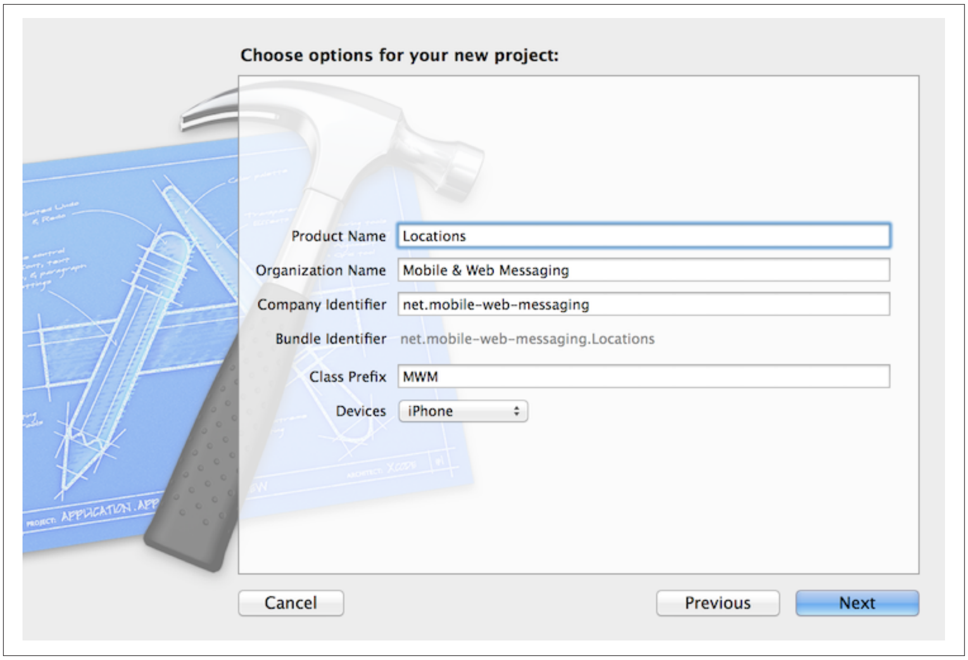
Create the Podfile

To import the library that we will use to send and receive messages, we will set up the project to use CocoaPods, an Objective-C Library Manager.

First, we need to close Xcode, because we will modify the project structure to import our dependencies.

After installing CocoaPods by following the instructions on its website, we create a file named *Podfile* at the root of the project (in the same directory as *Locations.xcodeproj*):

```
xcodeproj 'Locations.xcodeproj'

pod 'StompKit', '~> 0.1'

platform :ios, '5.0'
```


Choose options for your new project:

Product Name: Locations
Organization Name: Mobile & Web Messaging
Company Identifier: net.mobile-web-messaging
Bundle Identifier: net.mobile-web-messaging.Locations
Class Prefix: MWM
Devices: iPhone

Figure 2-4. XCode project options screen

After saving this file, run the **`pod install`** command:

```
$ pod install
Analyzing dependencies
Downloading dependencies
Installing CocoaAsyncSocket (7.3.2)
Installing StompKit (0.1.0)
Generating Pods project
Integrating client project

[!] From now on use `Locations.xcworkspace`.
```

We can now open Xcode again, but we must do it using the workspace file named *Locations.xcworkspace*, and not the project file named *Locations.xcodeproj* (Figure 2-5).

First, we will verify that the project is set up correctly and that the application can run in the iOS simulator.

We will simulate the latest iPhone devices by selecting Product → Destination → iPhone Retina (4-inch 64-bit) from the Xcode menu bar.

If we run the application by selecting Product → Run (or pressing ⌘+R), the iOS simulator starts and opens the application, which is composed of a blank view (Figure 2-6).

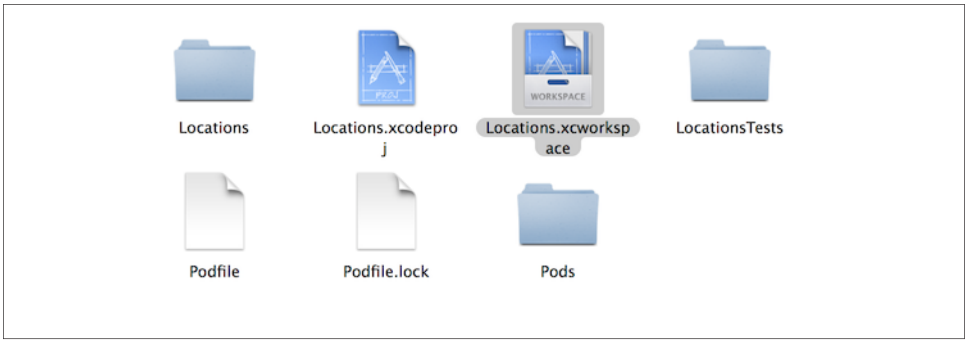


Figure 2-5. Open the Workspace file

Figure 2-6. Blank view of the Locations application

Nothing is displayed, but it confirms that the project and its dependencies are successfully compiled and launched.

Identify the Device

The iPhone device will broadcast its position. The first thing to do is identify the device. To keep the example simple, we will use a *universal unique identifier* (or UUID) as the device identifier and display it in the view.

Because the application will run only on iPhone devices, the entire user interface will be set up in the *Main.storyboard* file.

Click on *Main.storyboard* to open it. From the Object library, drag a Label on the View's window. Place it at the top of the view and change the text to "Device ID," as illustrated in Figure 2-7.

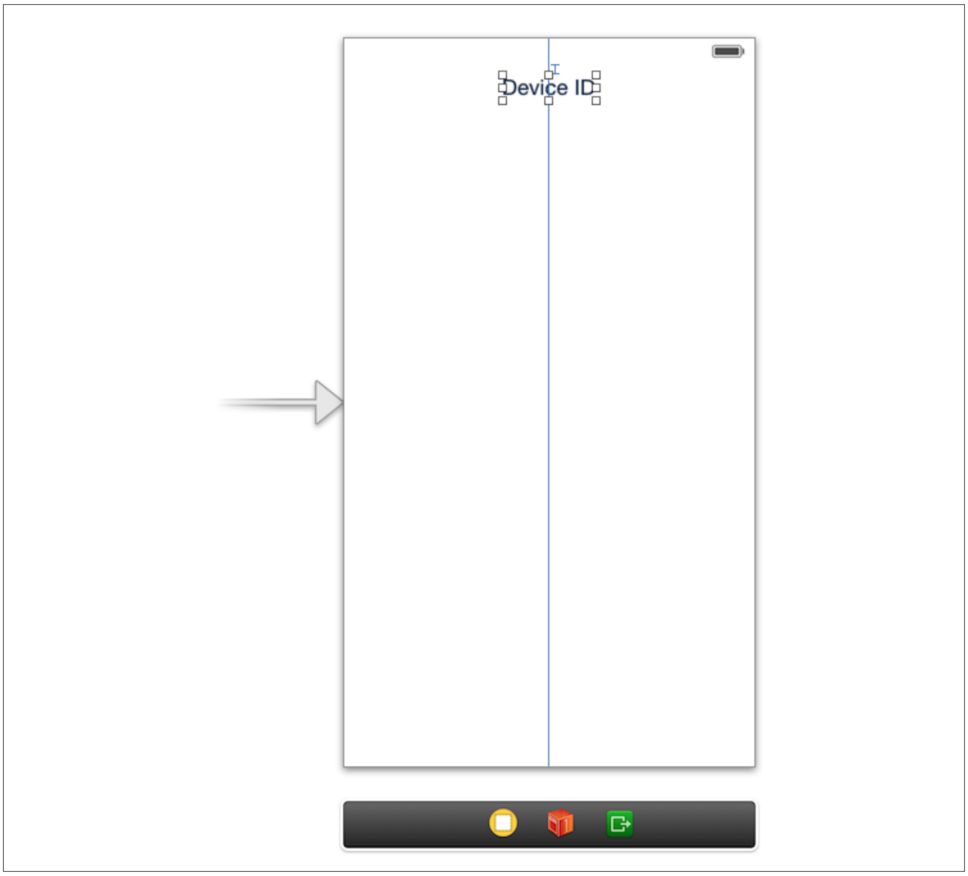


Figure 2-7. Add the Device ID label

I will not describe in detail how to set up the layout constraints for the graphical objects so that they adapt correctly to the device's size and orientation. However, the example code in the GitHub repository is constrained correctly.

The UUID that we will generate is quite long, so we will change its appearance by setting its Font to System 13.0 and its Alignment to centered to fit the screen, as illustrated in Figure 2-8.

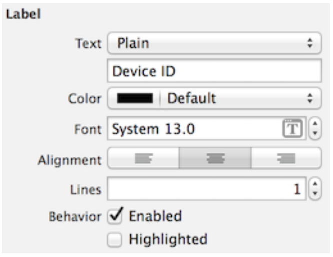

Label
Text: Plain
Device ID
Color: Default
Font: System 13.0
Alignment: centered
Lines: 1
Behavior: [✓ Enabled] Highlighted

Figure 2-8. Change the appearance of the device ID label

We will connect this label to the `MWMViewController` object.

Add the necessary outlet property in the `MWMViewController` private interface in *MWMViewController.m* and a `NSString` to hold the identifier:

```
@interface MWMViewController ()

@property (weak, nonatomic) IBOutlet UILabel *deviceIDLabel;

@property (copy, nonatomic) NSString *deviceID;

@end
```

Open the *Main.storyboard* and Ctrl-click on View Controller to see its connection panel. Drag from deviceIDLabel to the UILabel to connect it. See Figure 2-9.

Now that the outlet property is connected to the label, we need to generate a UUID for the application and display it when the view appears.

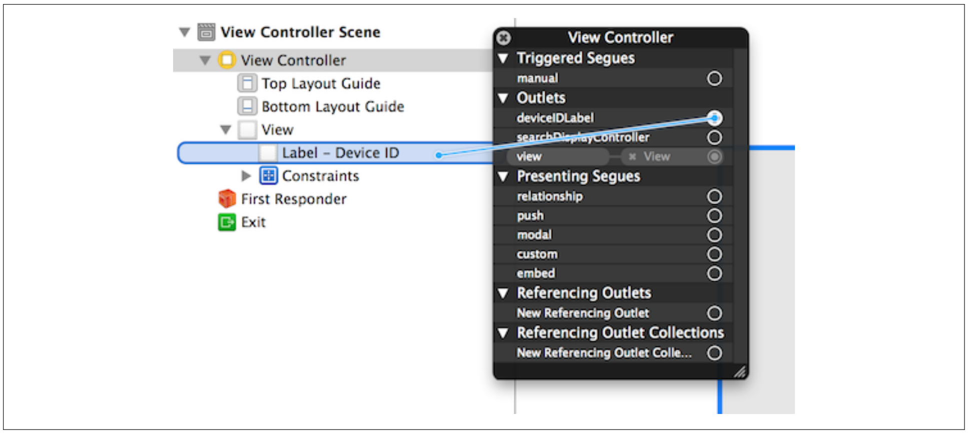


Figure 2-9. Connect the deviceIDLabel outlet property to the Device ID UILabel

Open the *MWMViewController.m* file to add code to the `MWMViewController` implementation. When the application starts and the view is loaded in `viewDidLoad`, we set the `deviceID` using a UUID:

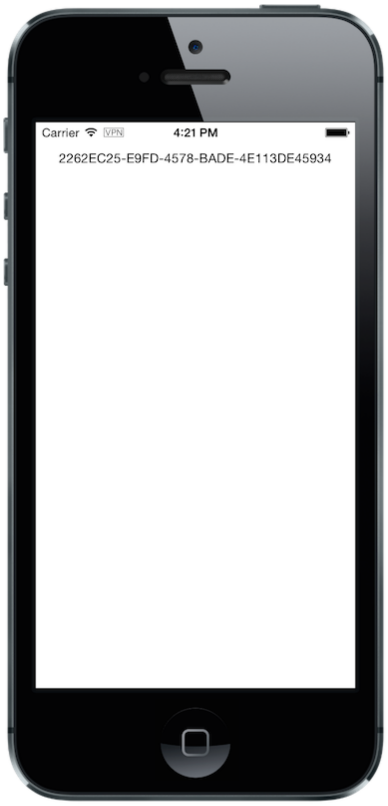
```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.deviceID = [UIDevice currentDevice].identifierForVendor.UUIDString;
    NSLog(@"Device identifier is %@", self.deviceID);
}
```

The `identifierForVendor` property will uniquely identify the device for the application's vendor (that we set to `net.mobile-web-messaging` when we created the project).

We also need to set the label to this ID when the view appears:

```
- (void)viewWillAppear:(BOOL)animated
{
    self.deviceIDLabel.text = self.deviceID;
}
```

If we run the application, we will see the device ID displayed instead of `Device ID` in the view (Figure 2-10).



2262EC25-E9FD-4578-BADE-4E113DE45934

Figure 2-10. Display the device ID

Now that we have the identifier of the device, the next step is to retrieve its geolocation data using the `CoreLocation` framework before we can send them in a STOMP message.

The following sections deal with setting up the framework and writing code to retrieve the GPS data from the device and display it. This is unrelated to messaging, and you can skip these sections if you only want to know how to send and receive messages. Still, I thought the messaging code would be more meaningful if it were using real data instead of generating random dummy data. By using GPS data instead, we will be able to build a mobile application that displays this data on a map in the next chapter.

Display the Device Position

We will retrieve the geolocation data from the device's GPS sensor to send them using STOMP messages. However, we also want to have some graphical feedback to show that the data changes over time as we move with our device.

To display the geolocation data, we will add a `UILabel` to the view and change its text to "Current position: ???" (Figure 2-11).

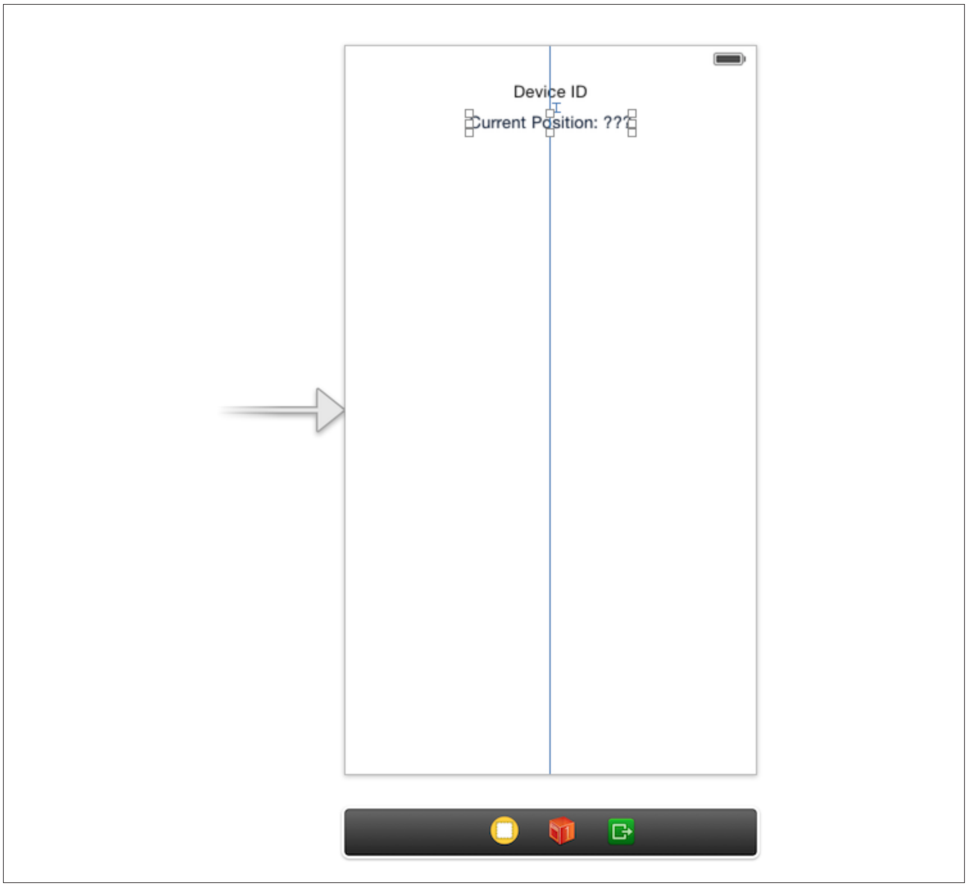
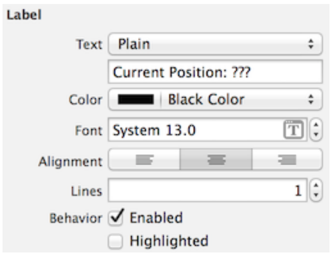


Figure 2-11. Add the current position label

We will change its appearance to match the `deviceID` label by setting its Font to System 13.0 and its Alignment to centered (Figure 2-12).


Label

Text: Plain

Current Position: ???

Color: Black Color

Font: System 13.0

Alignment

Lines: 1

Behavior: [✓ Enabled] Highlighted

Figure 2-12. Change the appearance of the current position label

Open the *MWMViewController.m* file and add a property to the `MWMViewController` private interface:

```
@property (weak, nonatomic) IBOutlet UILabel *currentPositionLabel;
```

We then bind this property to the label. Open the *Main.storyboard* and Ctrl-click on View Controller to see its connection panel. Drag from currentPositionLabel to the label to connect it. See Figure 2-13.

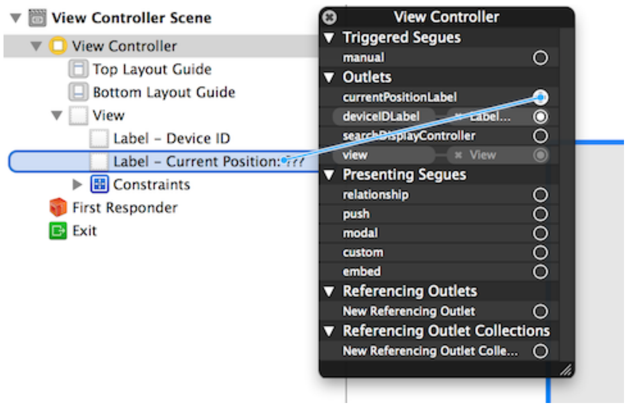


Figure 2-13. Connect the currentPositionLabel outlet property to the Current Position UILabel

The label is now connected to the property. The next step is to retrieve the geolocation data from the device to update this property and send a STOMP message with them.

Access the Device Geolocation Data with CoreLocation Framework

iOS provides the `CoreLocation` framework to access the location data.

We need to add it to the libraries linked by the application. Click on the Locations project and then the Locations target. In the General tab, under the Linked Frameworks and Libraries section, click on the + button. In the selection window, type **`Core Location`**, select the CoreLocation.framework, and click on the Add button (Figure 2-14).

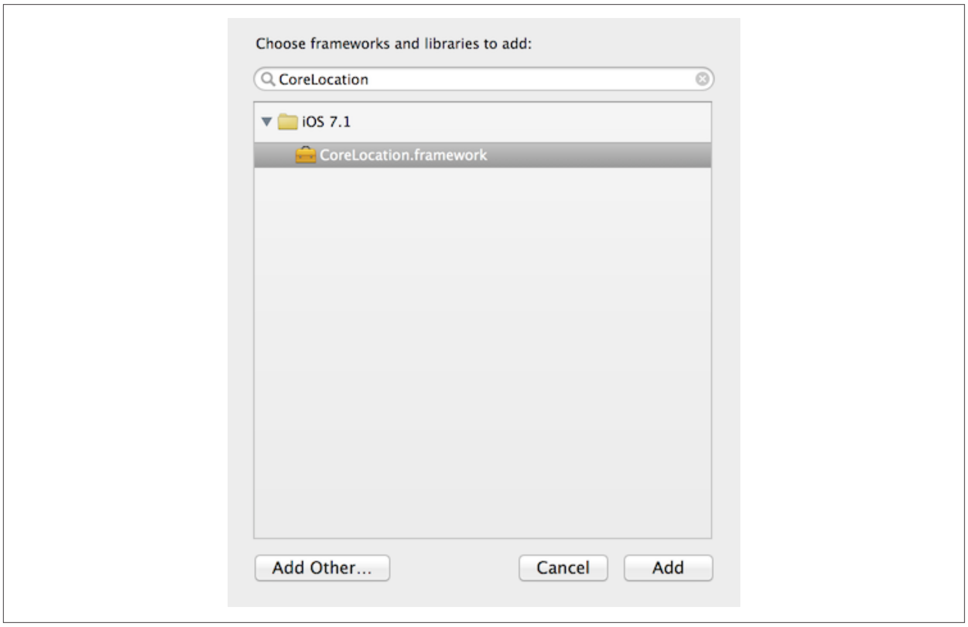


Figure 2-14. Add the CoreLocation framework

We can now use the CoreLocation framework by importing *CoreLocation/CoreLocation.h* at the top of the *MWMViewController.m* file.

We will make the `MWMViewController` private interface conform to the `CLLocationManagerDelegate` protocol and declare a `CLLocationManager` property named `locationManager`:

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

interface MWMViewController () <CLLocationManagerDelegate>

@property (strong, nonatomic) CLLocationManager *locationManager;

@end
```

We will define two methods to start and stop updating the current location. When the app starts updating the current location in `startUpdatingCurrentLocation`, it creates the `locationManager` if it is not already created and designates the controller as the `locationManager`'s `delegate`. Because the geolocation data will be used to follow the device as it moves, we set the locationManager's desiredAccuracy to kCLLocationAccuracyBestForNavigation.

Finally, the application will start listening for the device location by calling locationManager's startUpdatingLocation method:

```
#pragma mark - CoreLocation actions

- (void)startUpdatingCurrentLocation
{
    NSLog(@"startUpdatingCurrentLocation");

    // if location services are restricted do nothing
    CLAuthorizationStatus status = [CLLocationManager authorizationStatus];
    if (status == kCLAuthorizationStatusDenied ||
        status == kCLAuthorizationStatusRestricted) {
        return;
    }

    // if locationManager does not currently exist, create it
    if (!self.locationManager) {
        self.locationManager = [[CLLocationManager alloc] init];
        // set its delegate to self
        self.locationManager.delegate = self;
        // use the accuracy best suite for navigation
        self.locationManager.desiredAccuracy =
          kCLLocationAccuracyBestForNavigation;
    }

    // start updating the location
    [self.locationManager startUpdatingLocation];
}
```

To stop receiving the device location in stopUpdatingCurrentLocation, we simply call locationManager's stopUpdatingLocation method:

```
- (void)stopUpdatingCurrentLocation
{
    [self.locationManager stopUpdatingLocation];
}
```

The location of the device will be received by the designated CLLocationManagerDelegate (in our case, the MWMViewController implementation itself). We need to implement the locationManager:didUpdateToLocation:fromLocation: method and extract the coordinates from the newLocation's coordinate.

After we have them, we can update the currentPositionLabel's text to display them:

```
#pragma mark - CLLocationManagerDelegate protocol
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
```

```
            fromLocation:(CLLocation *)oldLocation
{
    // ignore if the location is older than 30s
    int delta = [newLocation.timestamp timeIntervalSinceDate: [NSDate date]];
    if (fabs(delta) > 30) {
        return;
    }

    CLLocationCoordinate2D coord = [newLocation coordinate];
    NSString *text =[NSString stringWithFormat:@"φ:%.4F, λ:%.4F",
                     coord.latitude, coord.longitude];
    self.currentPositionLabel.text = text;
}
```

If there is any problem with the `locationManager`, we want to warn the user about it and stop updating the location. To do so, we implement the `CLLocationManager Delegate`'s `locationManager:didFailWithError:` method to display a warning to the user:

```
- (void)locationManager:(CLLocationManager *)manager
       didFailWithError:(NSError *)error
{
    // reset the current position label
    self.currentPositionLabel.text = @"Current position: ???";

    // show the error alert
    UIAlertView *alert = [[UIAlertView alloc] init];
    alert.title = @"Error obtaining location";
    alert.message = [error localizedDescription];
    [alert addButtonWithTitle:@"OK"];
    [alert show];
}
```

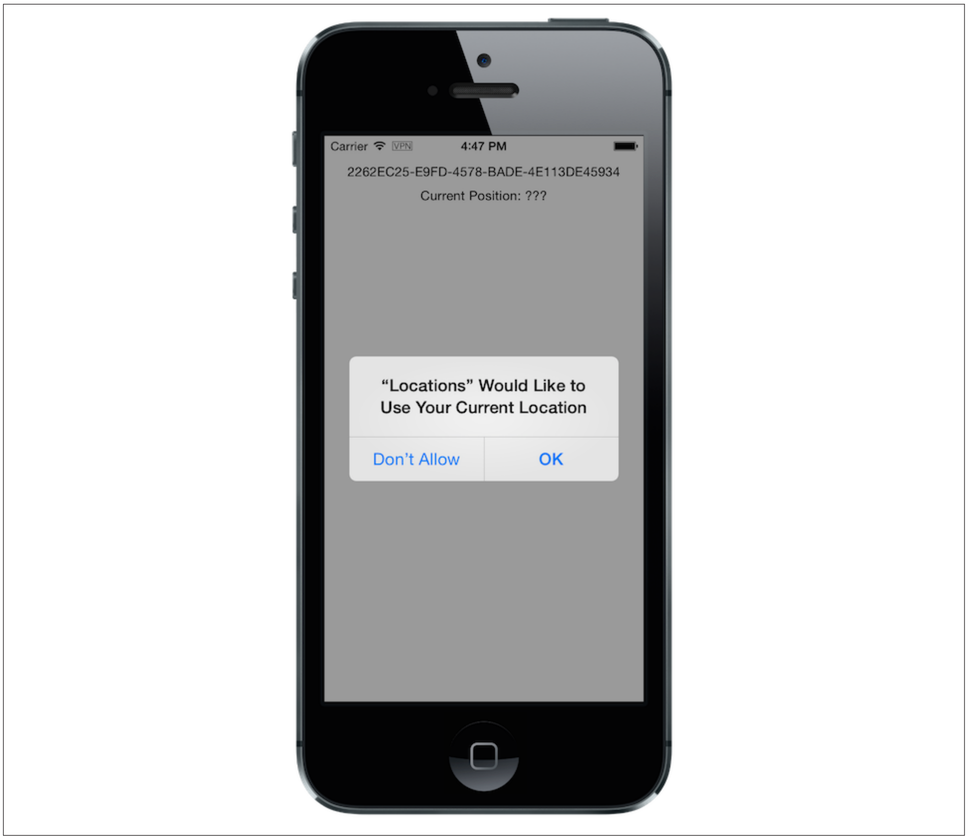
Now that the code related to `CoreLocation` is in place, we just need to call the `star tUpdatingCurrentLocation` method when the view appears:

```
- (void)viewWillAppear:(BOOL)animated
{
    self.truckIDLabel.text = self.truckID;

    [self startUpdatingCurrentLocation];
}
```

We also need to stop updating the location when the view disappears in `viewDidDi sappear:`.

```
- (void)viewDidDisappear:(BOOL)animated
{
    [self stopUpdatingCurrentLocation];
}
```

The first time the application asks the `locationManager` to start updating the device location, the user will see an alert view asking for permission to access the device location (Figure 2-15).



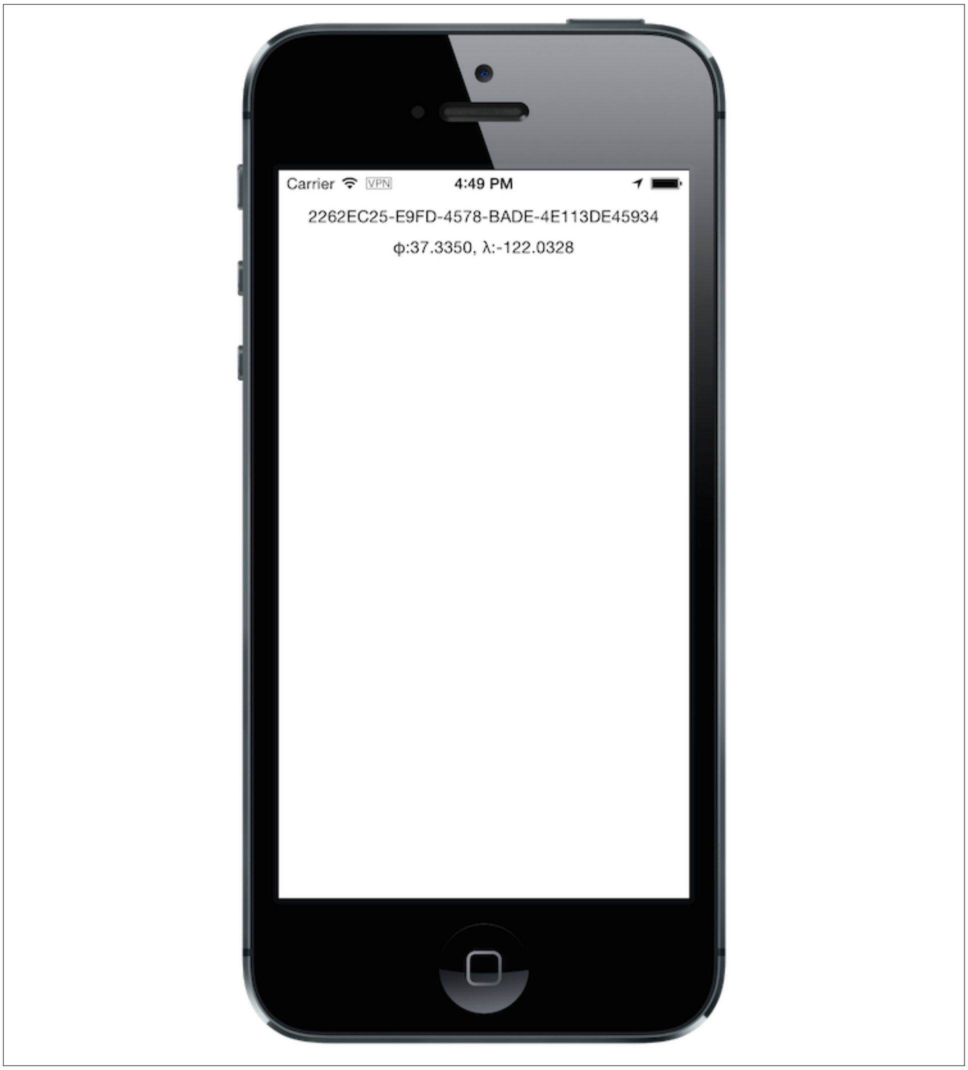
2262EC25-E9FD-4578-BADE-4E113DE45934

Current Position: ???

"Locations" Would Like to Use Your Current Location

Figure 2-15. Permission to use the current location

If the user taps OK, the `locationManager` will start updating the device location and the label for its current position will be updated with the latitude and longitude (Figure 2-16).



Carrier VPN 4:49 PM

2262EC25-E9FD-4578-BADE-4E113DE45934

ϕ:37.3350, λ:-122.0328

Figure 2-16. Display the current position of the device

Simulate a Location with iOS Simulator

If you are running the application on an iPhone device, the real geolocation data from the device will be used. If you run the application using the iOS Simulator, you can simulate different locations in the Debug → Location menu. For example, the Freeway Drive option will simulate a car driving on a freeway between Palo Alto and San Francisco.

Whether you are running the application on a device or in the simulator, you should see the currentPositionLabel be updated. The latitude and longitude numbers are difficult to interpret as such, but in Chapter 3 we will use them to draw the position on a map to locate the devices.

Now that the Locations application is handling the device geolocation data, the next step is to send them using STOMP.

Create a STOMP Client with StompKit

Before sending any messages, we must first import the StompKit library that we added to the *Podfile* file at the beginning of this chapter.

We must import its header file *StompKit.h* at the top of the *MWMViewController.m* file and add a STOMPClient property named client to the MWMViewController private interface:

```
#import <StompKit.h>

@interface MWMViewController () <CLLocationManagerDelegate>

@property (nonatomic, strong) STOMPClient *client;

@end
```

The client property will be used to communicate with the STOMP broker after it is created and connected.

We do not need to conform to any protocol to use StompKit, as its API is based on *blocks* instead of protocol delegates.

The client variable is created when the controller's view is loaded in MWMViewController's viewDidLoad method implementation. To create it, we need to pass the host and port of the STOMP broker to connect to. This information depends on the broker you are using. If you have configured ActiveMQ on your machine as described in Appendix A, you will be able to connect on its 61613 port.

The host will depend on your network configuration. On my local network, my server has the IP address *192.168.1.25*. I will use this value for the example, but you should replace this with your own server address to run the applications:

```
#define kHost     @"192.168.1.25"
#define kPort     61613

...

@implementation MWMViewController

- (void)viewDidLoad
```

```
{
    [super viewDidLoad];

    self.deviceID = [UIDevice currentDevice].identifierForVendor.UUIDString;
    NSLog(@"Device identifier is %@", self.deviceID);

    self.client = [[STOMPClient alloc] initWithHost:kHost port:kPort];
}
```

Connect to a STOMP Broker

When the `client` object is created, it is not yet connected to the STOMP broker. To connect, we must call its `connectWithHeaders:completionHandler:` method.

StompKit uses Grand Central Dispatch and blocks to provide an event-driven API. This means that the client is *not* connected when the call to its `connectWithHeaders:completionHandler:` method returns but when the `completionHandler` block is called.

We can pass a dictionary to `connectWithHeaders:completionHandler:` to add additional headers during the connection to the STOMP broker. In our application, we will send a `client-id` header set to the `deviceID` to uniquely identify the client against the STOMP broker.

This ensures that no two devices will be able to connect using the same identifier. After a client is connected with a given `client-id`, any subsequent clients that use the same value will fail to connect to the broker.

We will encapsulate this code in a `connect` method in the `MWMViewController` implementation.

```
@implementation MWMViewController

#pragma mark - Messaging
- (void)connect
{
    NSLog(@"Connecting...");
    [self.client connectWithHeaders:@{ @"client-id": self.deviceID }
                  completionHandler:^(STOMPFrame *connectedFrame,
                                      NSError *error) {
                      if (error) {
                          // We have not been able to connect to the broker.
                          // Let's log the error
                          NSLog(@"Error during connection: %@", error);
                      } else {
                          // we are connected to the STOMP broker without
                          // an error
                          NSLog(@"Connected");
                      }
                  }];
```

```
        // when the method returns, we can not assume that the client is connected
    }


    @end
```

We will call this `connect` method when the view appears in `viewWillAppear:`.

```
- (void)viewWillAppear:(BOOL)animated
{
    self.truckIDLabel.text = self.truckID;

    [self startUpdatingCurrentLocation];
    [self connect];
}
```

Disconnect from a STOMP Broker

The `STOMPClient` disconnects from the broker using its `disconnect:` method. This method takes a block that will be called when the client is disconnected from the server. The block takes an `NSError` parameter that is set if there is an error during the disconnection operation.

```
#pragma mark - Messaging

- (void)disconnect
{
    NSLog(@"Disconnecting...");
    [self.client disconnect:^(NSError *error) {
        if (error) {
            NSLog(@"Error during disconnection: %@", error);
        } else {
            // the client is disconnected from the broker without any problem
            NSLog(@"Disconnected");
        }
    }];
    // when the method returns, we cannot assume that the client is disconnected
}
```

We will disconnect from the broker after the view has disappeared in `viewDidDisappear:`.

```
- (void)viewDidDisappear:(BOOL)animated
{
    [self stopUpdatingCurrentLocation];
    [self disconnect];
}
```

At this stage, we have an application that connects to the STOMP broker when its view is displayed and disconnects when its view disappears.

If we run the application, we see logs in Xcode that show the connection process:

```
2014-03-13 17:07:21.667 Locations[79069:60b] Connecting...
2014-03-13 17:07:21.723 Locations[79069:3903] Connected
```

Send STOMP Messages

We now have a connection to the STOMP broker and receive the device's geolocation data from the `CoreLocation` framework. The last step is to send this data to the topic associated to the device ID.

As we described in "Messaging models for the Locations application" on page 6, each device will send its location in a topic named after its identifier.

```
NSString *destination = [NSString
 stringWithFormat:@"/topic/device.
 %@.location",
 self.deviceID];
```

ActiveMQ convention is to prefix a STOMP destination with `/topic/` to use a publish/subscribe messaging model and by `/queue/` to use a point-to-point model.

We designed our application to use a topic for the `device.XXX.location`, so we must prefix it with `/topic/`.

As we described in "Message representation for the Locations application" on page 7, the message representation is a JSON string that contains the location coordinates, the timestamp, and the device ID. We build an `NSDictionary` from this data and serialize it as a JSON string:

```
// build a dictionary containing all the information to send
NSDictionary *dict = @{
    @"deviceID": self.deviceID,
    @"lat": [NSNumber numberWithDouble:location.coordinate.latitude],
    @"lng": [NSNumber numberWithDouble:location.coordinate.longitude],
    @"ts": [dateFormatter stringFromDate:location.timestamp]
};
// create a JSON string from this dictionary
NSData *data = [NSJSONSerialization dataWithJSONObject:dict
                                               options:0
                                                 error:nil];
NSString *body =[[NSString alloc] initWithData:data
                                      encoding:NSUTF8StringEncoding];
```

This body follows the JSON format. We will add a `content-type` header in the STOMP message and set it to `application/json; charset=utf-8` to let the STOMP brokers and the eventual consumers know that this message's payload can be read as a JSON string encoded with UTF-8. Without such a `content-type`, the consumers would not necessarily know how to *read* the data in the body and interpret it.

```
NSDictionary *headers = @{
    @"content-type": @"application/json;charset=utf-8"
};
```

We now have the `destination`, `headers`, and `body` to send in the message. The last step is to use the `client`'s `sendTo:headers:body` method to send it.

```
// send the message
[self.client sendTo:destination
            headers:headers
               body:body];
```

We will encapsulate all of these steps in a `sendLocation:` method that takes a `CLLocation` parameter.

```
- (void)sendLocation:(CLLocation *)location
{
    // build a static NSDateFormatter to display the current date in ISO-8601
    static NSDateFormatter *dateFormatter = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        dateFormatter = [[NSDateFormatter alloc] init];
        dateFormatter.dateFormat = @"yyyy-MM-d'T'HH:mm:ssZZZZZ";
    });

    // send the message to the device's topic
    NSString *destination =
        [NSString stringWithFormat:@"/topic/device.%@.location", self.deviceID];

    // build a dictionary containing all the information to send
    NSDictionary *dict = @{
        @"deviceID": self.deviceID,
        @"lat": [NSNumber numberWithDouble:location.coordinate.latitude],
        @"lng": [NSNumber numberWithDouble:location.coordinate.longitude],
        @"ts": [dateFormatter stringFromDate:location.timestamp]
    };
    // create a JSON string from this dictionary
    NSData *data = [NSJSONSerialization dataWithJSONObject:dict
                                                   options:0
                                                     error:nil];
    NSString *body =[[NSString alloc] initWithData:data
                                          encoding:NSUTF8StringEncoding];

    NSDictionary *headers = @{
        @"content-type": @"application/json;charset=utf-8"
    };

    // send the message
    [self.client sendTo:destination
                headers:headers
                   body:body];
}
```

The next step is to call this method every time we receive an updated location in the `locationManager:didUpdateToLocation:fromLocation:` method.

```
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
           fromLocation:(CLLocation *)oldLocation
{
    // ignore if the location is older than 30s
    int delta = [newLocation.timestamp timeIntervalSinceDate:[NSDate date]];
    if (fabs(delta) > 30) {
        return;
    }

    CLLocationCoordinate2D coord = [newLocation coordinate];
    NSString *text =[NSString stringWithFormat:@"φ:%.4F, λ:%.4F",
                     coord.latitude, coord.longitude];
    self.currentPositionLabel.text = text;

    // send a message with the location data
    [self sendLocation:newLocation];
}
```

Messages will be sent every time the device location changes. This is a bit inconvenient during development, as I do not want to move around my workspace whenever I need to update my location and send a message.

To simplify the development process, we will add code to send the last known location when the user shakes the device.

We will need to add a `lastKnownLocation` property to the `MWMViewController` private interface and use it to store the location returned by the `CLLocationManager` delegate method:

```
@interface MWMViewController () <CLLocationManagerDelegate>

...
@property (strong, nonatomic) CLLocation *lastKnownLocation;

@end

@implementation MWMViewController

...

#pragma mark - CLLocationManagerDelegate protocol

- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
           fromLocation:(CLLocation *)oldLocation
{

    ...
```

```
        // send a message with the location data
        [self sendLocation:newLocation];
        // store the location to send it again when user shakes the device
        self.lastKnownLocation = newLocation;
    }
```

To send a message with this `lastKnownLocation` when the user shakes the device, we must implement the `motionEnded:withEvent:` method in the `MWMViewController` implementation and check if the event is a motion shake (identified by `UIEventSubtypeMotionShake`):

```
    #pragma mark - User Events

    - (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
    {
        if (motion == UIEventSubtypeMotionShake) {
            NSLog(@"device is shaked");
            if (self.lastKnownLocation) {
                [self sendLocation:self.lastKnownLocation];
            }
        }
    }
```

When we run the application, a STOMP message will be sent every time the location manager updates the device's location or when the user shakes the device.

How can we check that messages are effectively sent?

We will confirm it at three different stages:

1. We will display the debug log on the device to check that messages are sent.
2. Next, we'll use the ActiveMQ administration console to check that it effectively handled the sent messages.
3. Finally, we will write the simplest STOMP consumer that can receive these messages.

Display StompKit Debug Log

Every time the StompKit library sends a message to a STOMP broker, it logs the STOMP frame that is sent.

To display them in the console, edit the file named *StompKit.m* in Xcode that is under the *Pods* project (its full path is Pods → Pods → StompKit → StompKit.m in the Project Navigator view) and change the macro to activate logs by replacing the `0` with `1`:

```
#pragma mark Logging macros

#if 1 // set to 1 to enable logs

...
```

If we restart the application, we now see debug statements in Xcode's Debug console:

```
2014-03-13 17:19:05.711 Locations[79549:60b] >>> SEND
destination:/topic/device.2262EC25-E9FD-4578-BADE-4E113DE45934.location
content-type:application/json;charset=utf-8
content-length:122

{"lng":-122.03254905,"deviceID":"2262EC25-E9FD-4578-BADE-4E113DE45934","lat":
37.33521504,"ts":"2014-03-13T17:19:05+01:00"}
...
```

This confirms that STOMP messages are effectively sent by the Locations application.

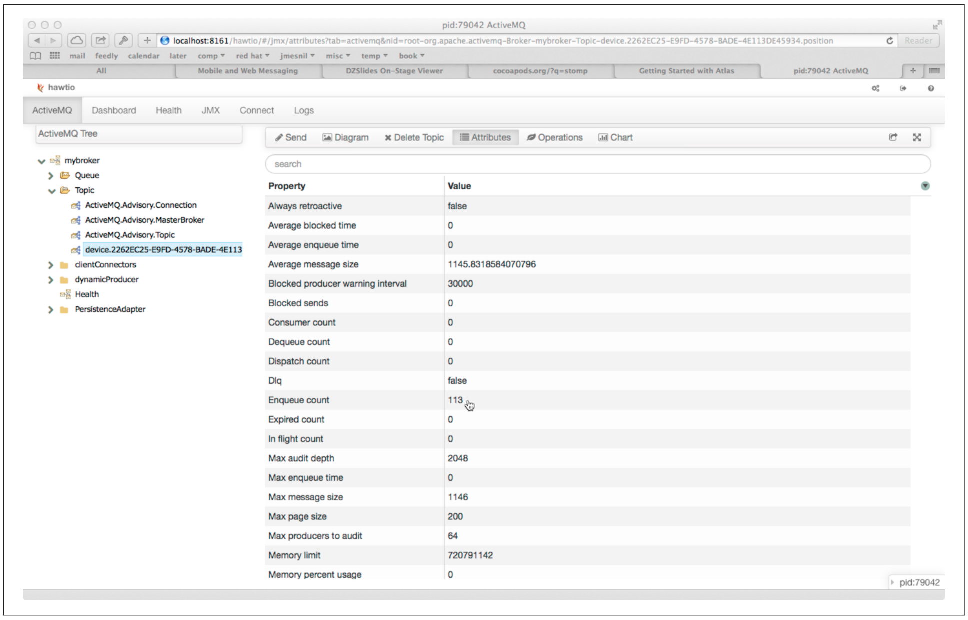
ActiveMQ Admin Console

In app_activemq_admin_console, we use the ActiveMQ admin console to check the broker configuration. We can also use this console to check the destinations and their associated metrics.

Go to the ActiveMQ admin console in your web browser at *http://localhost:8161/hawtio* and navigate the ActiveMQ tree down to the position topic in mybroker → Topic → device.2262EC25-E9FD-4578-BADE-4E113DE45934.location.

In the righthand panel, select Attributes in the top menu to display all the attributes associated to this topic.

To check whether the broker is receiving the messages on this destination, check the Enqueue count attribute. It corresponds to the messages that have been *enqueued* (in other words, *sent*) to the destination. We see that this value is growing over time (it was at 113 when the screenshot in Figure 2-17 was captured). This confirms that the broker is actually receiving the messages sent by the mobile application.


pid:79042 ActiveMQ
localhost:8161/hawtio/#/jmx/attributes?tab=activemq&nid=root-org.apache.activemq-Broker-mybroker-Topic-device.2262EC25-E9FD-4578-BADE-4E113DE45934.position
hawtio
ActiveMQ
Dashboard
Health
JMX
Connect
Logs
ActiveMQ Tree
mybroker
Queue
Topic
ActiveMQ.Advisory.Connection
ActiveMQ.Advisory.MasterBroker
ActiveMQ.Advisory.Topic
device.2262EC25-E9FD-4578-BADE-4E113
clientConnectors
dynamicProducer
Health
PersistenceAdapter
Send
Diagram
Delete Topic
Attributes
Operations
Chart
search

Property	Value
Always retroactive	false
Average blocked time	0
Average enqueue time	0
Average message size	1145.8318584070796
Blocked producer warning interval	30000
Blocked sends	0
Consumer count	0
Dequeue count	0
Dispatch count	0
Dlq	false
Enqueue count	113
Expired count	0
In flight count	0
Max audit depth	2048
Max enqueue time	0
Max message size	1146
Max page size	200
Max producers to audit	64
Memory limit	720791142
Memory percent usage	0

Figure 2-17. Check the number of messages sent to a destination in the ActiveMQ admin console

Another interesting attribute is Dequeue count. It corresponds to the messages removed from the topic and sent to consumers. In our case, it stays at `0` because there are no consumers subscribed to this destination.

A Simple STOMP Consumer

When I presented STOMP, I wrote that the protocol is so simple that a telnet client *is* a STOMP client.

Let's prove this by writing the simplest STOMP client that will consume the messages sent by the application to the destination.

We need to open a telnet client to connect to the broker host on the `61613` port. I am on the same machine as the broker, so I will simply connect to `localhost`:

```
$ telnet localhost 61613
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

Once the client is connected, we must connect to the broker to open a STOMP connection (as we did in the application using `STOMPClient`'s `connectWithHeaders:completionHandler:` method):

```
CONNECT

^@
```

A STOMP frame must be ended with a NULL octet.

`^@` is the ASCII character for a NULL octet. Type Ctrl + @ to enter it.

Note also that there is a blank line between the `CONNECT` line and the NULL octet. This blank line is mandatory to separate the command name and the headers from the beginning of the optional payload (which is not present in the `CONNECT` frame).

Once you type Ctrl + @, the messaging broker will process the `CONNECT` frame and reply with a `CONNECTED` frame:

```
CONNECTED
heart-beat:0,0
session:ID:jeff.local-63055-1391518653216-2:23
server:ActiveMQ/5.9.0
version:1.2
```

The STOMP connection is now established and the telnet client can now exchange messages with the broker. We are only interested in consuming messages sent by the application on the device's location topic. The device ID is displayed on the application screen. You will have to adapt the command to use your own device ID to receive its message:

```
SUBSCRIBE
destination:/topic/device.2262EC25-E9FD-4578-BADE-4E113DE45934.location

^@
```

After sending this command to the STOMP broker, we will immediately receive `MESSAGE` frames that correspond to the messages sent by the application:

```
MESSAGE
content-type:application/json;charset=utf-8
message-id:ID:jeff.local-50971-1394726830317-2:5:-1:1:323
destination:/topic/device.2262EC25-E9FD-4578-BADE-4E113DE45934.location
timestamp:1394727930755
expires:0
content-length:122
priority:4

{"lng":-122.12966111,"deviceID":"2262EC25-E9FD-4578-BADE-4E113DE45934","lat":
37.36492641,"ts":"2014-03-13T17:25:30+01:00"}
```

We can see that there are more headers in the consumed messages than in the messages we sent (which only had `content-type` and `content-length`). These headers are added by the STOMP broker and provide additional metadata about the messages. We will explore some of them later in Chapter 4 and Chapter 5.

At this stage, we have a mobile application that is a STOMP *producer*. It broadcasts its position by sending messages to a STOMP destination.

Display the Text Messages

We will now write the second part of the Locations application (which will *consume* STOMP messages containing some text and display them in a table). We will write the graphical part first by adding a `UITable` to the user interface. Click on *Main.storyboard* to open it. From the Object library, drag a Table View on the View's window. Place it below the Current Position UILabel (see Figure 2-18).

From the Object library, drag a Table View Cell inside the Table View (see Figure 2-19). We will change the Table View Cell properties by setting its Style to Basic and its Identifier to TextCell (Figure 2-20).

The `MWMViewController` interface will be declared as both the data source and delegate of the table. Open the *MWMViewController.m* file, make the `MWMViewController` interface conform to the `UITableViewDataSource` and `UITableViewDelegate` protocols and add an outlet property for the table.

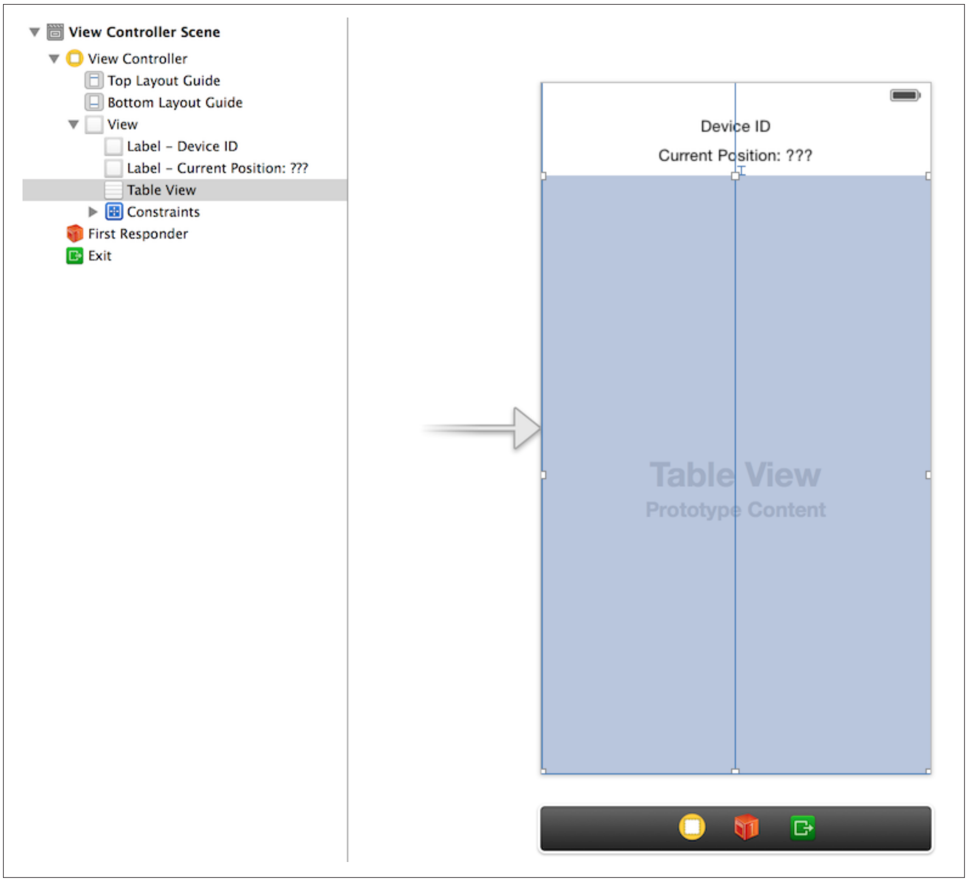


Figure 2-18. Add a Table View

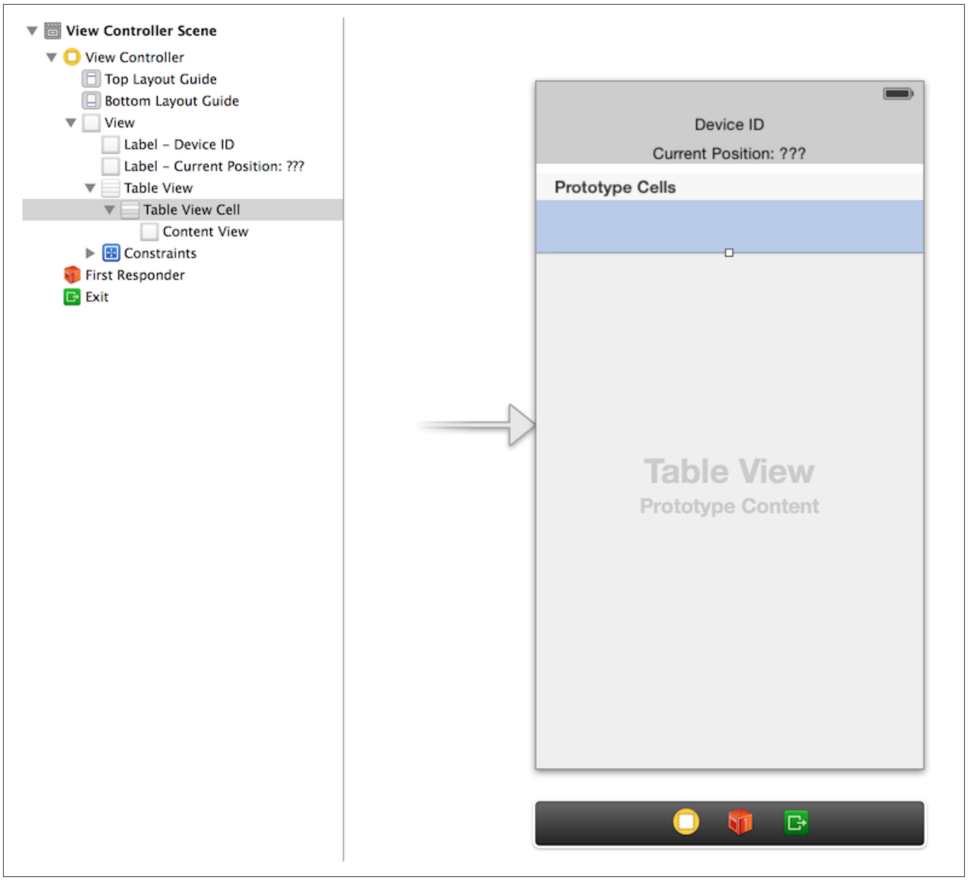


Figure 2-19. Add a Table View Cell

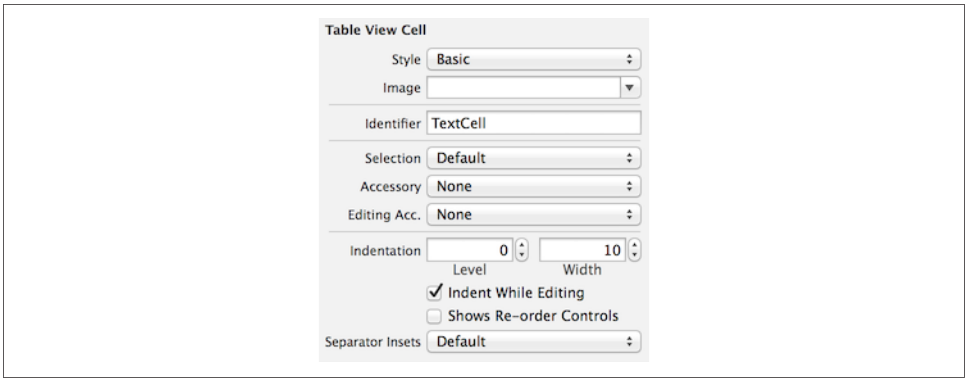


Table View Cell

Style	Basic
Image	
Identifier	TextCell
Selection	Default
Accessory	None
Editing Acc.	None
Indentation	0 (Level) 10 (Width)
	✓ Indent While Editing
	Shows Re-order Controls
Separator Insets	Default

Figure 2-20. Edit the Table View Cell properties

```
@interface MWMViewController () <CLLocationManagerDelegate, UITableViewData
Source, UITableViewDelegate>

@property (weak, nonatomic) IBOutlet UITableView *tableView;

@end
```

We need to bind this outlet property to the table view. Open the *Main.storyboard* and Ctrl-click on View Controller to see its connection panel. Drag from Table View the table to connect it. See Figure 2-21.

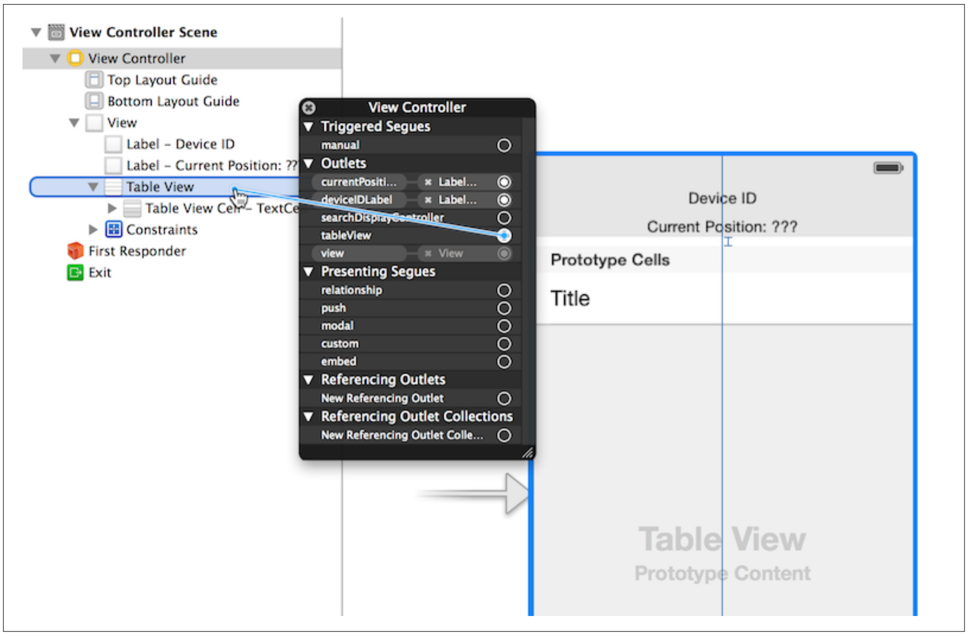


Figure 2-21. Connect the tableView outlet property to the Table View

We also need to connect the View Controller to the Table View and declare it as its dataSource and delegate.

Open the *Main.storyboard* and Ctrl-click on Table View to see its connection panel. Drag from dataSource to the View Controller to connect it (Figure 2-22).

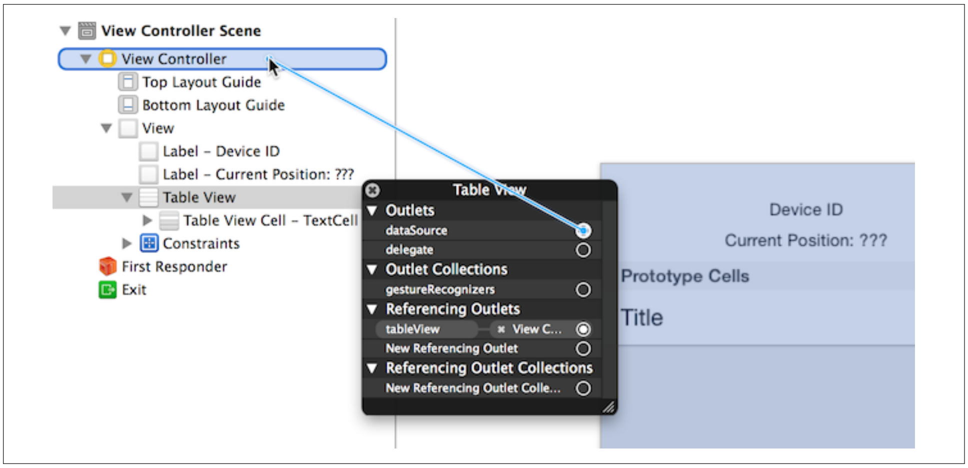


Figure 2-22. Connect the Table View's dataSource to the View Controller

We do the same operation to connect the Table View's delegate property to the View Controller (Figure 2-23).

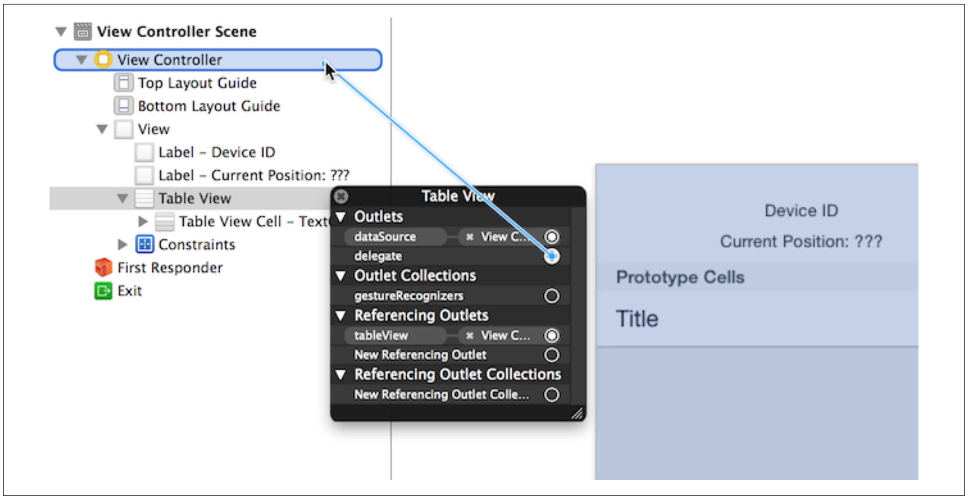


Figure 2-23. Connect the Table View's delegate property to the View Controller

The graphical objects are now properly connected to the properties. The next step is to make the `MWMViewController` implementation comply with the `UITableViewData Source` and `UITableViewDelegate` protocols.

The table will only display the received text messages. As there is no interaction with the table, we do not need to add any methods from the `UITableViewDelegate` protocol. Let's just add a comment to the `MWMViewController` implementation to remember this:

```
#pragma mark - UITableViewDelegate

// no delegate actions
```

The controller is also the `dataSource` of the table. We will keep a list of the texts in memory in an array. Let's add a `texts` array to the `MWMViewController` implementation and instantiate it in its `viewDidLoad` method:

```
@implementation MWMViewController

// the texts are stored in an array of NSString
NSMutableArray *texts;

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.deviceID = [UIDevice currentDevice].identifierForVendor.UUIDString;
    NSLog(@"Device identifier is %@", self.deviceID);

    self.client = [[STOMPClient alloc] initWithHost:kHost port:kPort];

    texts = [[NSMutableArray alloc] init];
}
```

This `texts` array will be used as the source of data for the table. Let's implement the required `UITableViewDataSource` methods:

```
#pragma mark - UITableViewDataSource protocol

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return [texts count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // this identifier must be the same as the onethat was set in
    // the Table View Cell properties in the storyboard
    static NSString *CellIdentifier = @"TextCell";

    UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:CellIdentifier];

    cell.textLabel.text = [texts objectAtIndex:indexPath.row];
    return cell;
}
```

With these methods implemented, the table will display all the texts that are stored in the `texts` array.

Receive STOMP Messages

Now that we are ready to display the table, the next step is to subscribe to the device's text destination to consume STOMP messages and store their text payload in the `texts` array.

To consume messages, a STOMP client must:

1. Connect to the broker.
2. Subscribe to the destination from which it wants to consume messages.

Subscribe to a STOMP Destination

We already took care of the first step by calling `STOMPClient`'s `connectWithHeaders:completionHandler:` in `MWMViewController`'s `connect` method.

Step 2 is handled in StompKit by calling `STOMPClient`'s `subscribeTo:headers:messageHandler:` method.

This method takes three parameters:

- The `destination` that the client wants to consume from. In our case, it is the destination `/queue/device.XXX.text` (we prepended the destination with the `/queue/` prefix according to the ActiveMQ naming convention).
- A dictionary of `headers` to pass additional metadata to the connection process. Because we do not have any such header for the time being, we will pass an empty dictionary.
- A `STOMPMessageHandler` block with a `STOMPMessage` parameter that will be called every time the broker sends a messages to the client to consume it. In our case, we just have to get the `NSString` text from the message `body` property and add it to the `texts` array.

We will add a method named `subscribe` to the `MWMViewController` implementation:

```
#pragma mark - Messaging

- (void)subscribe
{
    // susbscribes to the device text queue:
    NSString *destination =
        [NSString stringWithFormat:@"/queue/device.%@.text", self.deviceID];
    NSLog(@"subscribing to %@", destination);
    subscription = [self.client subscribeTo:destination
                                    headers:@{}
                             messageHandler:^(STOMPMessage *message) {
        // called every time a message is consumed from the destination
        NSLog(@"received message %@", message);
        // the text is send in a plain String, we use it as is.
        NSString *text = message.body;
        NSLog(@"adding text = %@", text);
        [texts addObject:text];
        // TODO reloads the table
    }];
}
```

subscription is an object returned by the subscribe method, which identifies the STOMP subscription and that can be used to *unsubscribe*.

We declare this object in the MWMViewController's implementation:

```
@implementation MWMViewController

STOMPSubscription *subscription;
```

We need to call this subscribe method as soon as the client is connected to the STOMP broker. The correct location is inside the completionHandler block of the connect method that will be called when the client is *successfully* connected to the STOMP broker:

```
#pragma mark - Messaging

- (void)connect
{
    NSLog(@"Connecting...");
    [self.client connectWithHeaders:@{ @"client-id": self.deviceID }
              completionHandler:^(STOMPFrame *connectedFrame,
                                  NSError *error) {
                  if (error) {
                      // We have not been able to connect to the broker.
                      // Let's log the error
                      NSLog(@"Error during connection: %@", error);
                      [weakSelf reconnect:error];
                  } else {
                      // we are connected to the STOMP broker without
                      // an error
                      NSLog(@"Connected");
                      [self subscribe];
                  }
              }];
    // when the method returns, we can not assume that the client is connected
}
```

Unsubscribe from the Destination

The application will consume messages from the destination as long as it remains connected to the STOMP broker.

We do not need to explicitly unsubscribe from the destination when we disconnect from the broker, but it is a good practice to do so. To unsubscribe, we just need to call the `unsubscribe` method on the `subscription` object that was created when we subscribed to the text destination. We will unsubscribe just prior to disconnecting from the broker in the `viewDidDisappear:` method:

```
- (void)viewDidDisappear:(BOOL)animated
{
    [self stopUpdatingCurrentLocation];
    [subscription unsubscribe];
    [self disconnect];
}
```

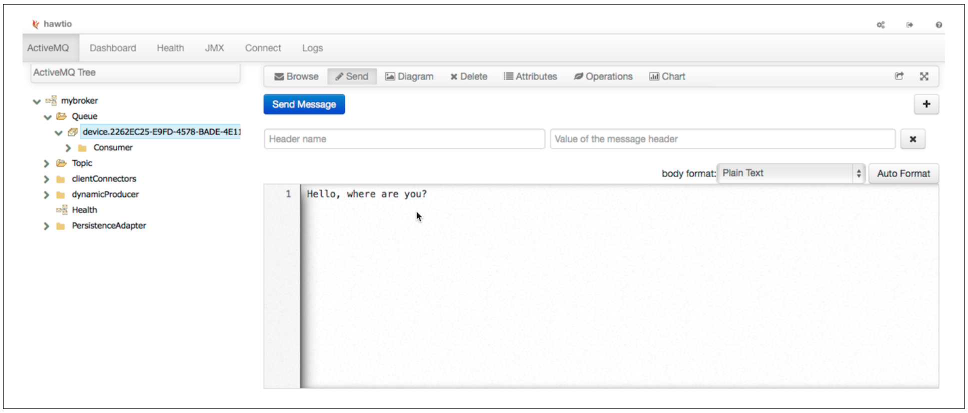
Finish the Application

The application is now ready to consume messages. Let's start it and check that it is working.

Run the application in the iOS simulator or on your device.

Go to the ActiveMQ admin console and browse to the device text destination (in my case, its name is device.2262EC25-E9FD-4578-BADE-4E113DE45934.text) and click on the Send tab.

As illustrated in Figure 2-24, fill the text area with a plain-text string (e.g., "Hello, where are you?") and set the body format to Plain Text.



body format: Plain Text

```
Hello, where are you?
```

Figure 2-24. Send a message using the ActiveMQ Admin Console

Click on the Send Message button to send the message on the destination.

We see in the application log that a STOMP message has been received and that the text was extracted from the message's body:

```
2014-03-14 14:24:19.807 Locations[86050:3903] received message MESSAGE
priority:0
destination:/queue/device.2262EC25-E9FD-4578-BADE-4E113DE45934.text
timestamp:1394803459806
message-id:ID\cjeff.local-53346-1394795959634-37\c1\c1\c1\c1
expires:0
subscription:sub-0

Hello, where are you?
2014-03-14 14:24:19.808 Locations[86050:3903] adding text = Hello, where are
you?
```

However, nothing is displayed in the application. We forgot to reload the table to display the received orders.

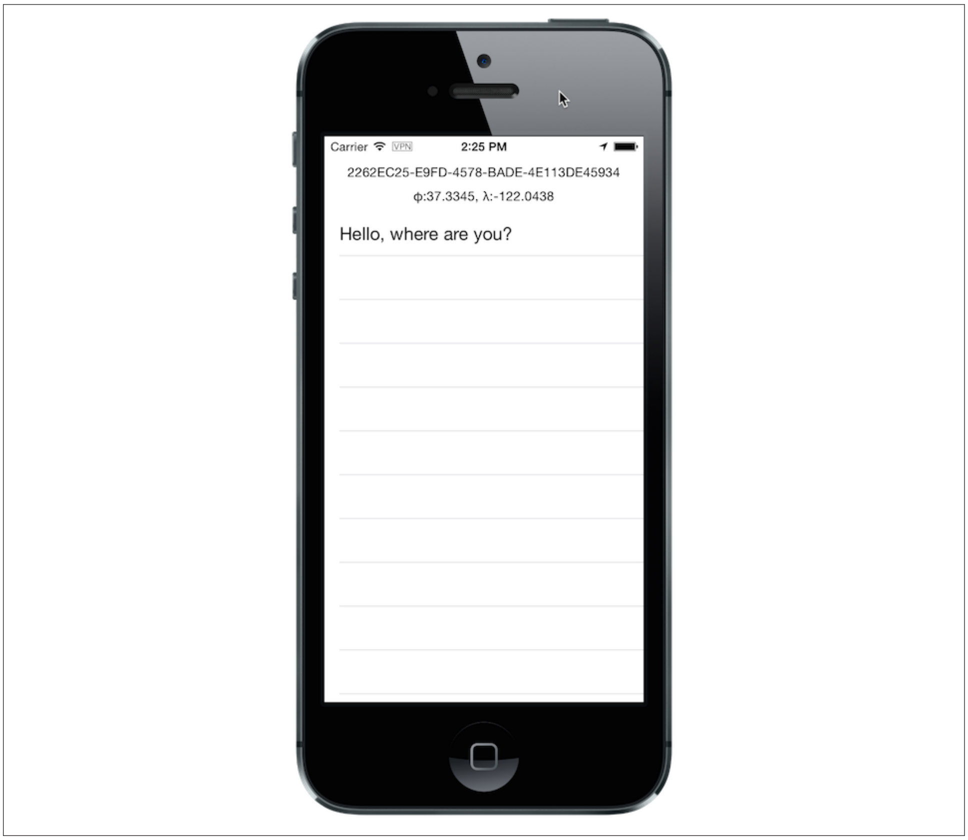
Let's fix that by calling reloadData on the tableView property from the STOMPMessageHandler block:

```
- (void)subscribe
{
    // subscribes to the device text queue:
    NSString *destination =
        [NSString stringWithFormat:@"/queue/device.%@.text", self.deviceID];

    NSLog(@"subscribing to %@", destination);
    subscription = [self.client subscribeTo:destination
                                    headers:@{}
                             messageHandler:^(STOMPMessage *message) {
        // called every time a message is consumed from the destination
        NSLog(@"received message %@", message);
        // the text is send in a plain String, we use it as is.
        NSString *text = message.body;
        NSLog(@"adding text = %@", text);
        [texts addObject:text];
        dispatch_async(dispatch_get_main_queue(), ^{
            [self.tableView reloadData];
        });
    }];
}
```

Note that we did not call directly [self.tableView reloadData]; from the STOMPMessageHandler block.

StompKit uses Grand Central Dispatch's global queue to handle the communication between the client and the STOMP brokers. The STOMPMessageHandler block is called on that queue. However, any code that deals with UIKit (such as reloading the `tableView`) *must* be executed on the queue bound to the main thread. This is why we must wrap the `reloadData` call into a block executed on the main queue.

If we restart the application and send another message on the destination with the ActiveMQ admin console, the table will display the text as soon as it is received. See Figure 2-25.


Carrier VPN 2:25 PM

2262EC25-E9FD-4578-BADE-4E113DE45934

ϕ:37.3345, λ:-122.0438

Hello, where are you?

Figure 2-25. The received text is displayed in the table

Summary

In this chapter, we learned to use StompKit to send and receive STOMP messages from an iOS application.

To send a message, the application must do the following:

1. Connect to the STOMP broker.
2. Send the message to the destination.

To consume a message, the application must do the following:

1. Connect to the STOMP broker.
2. Subscribe to the destination and pass a block that is called every time a message is received. This block is executed on the Grand Central Dispatch global queue. If there is any code that changes the user interface, it must be wrapped in a block executed on the main queue.

We use two different types of message payloads:

- A JSON payload by using its string representation for the message body and specifying `application/json; charset=utf-8` in its `content-type` header
- A simple plain-text payload without any `content-type` header

Sending and consuming messages is only possible when the client is *successfully* connected to the STOMP broker. Due to the event-driven design of StompKit, this is the case when the `completionHandler` block is executed without an error in `connectWithHeaders:completionHandler:`.

CHAPTER 3

Web Messaging with STOMP

In this chapter, we will write a web application that sends and receives messages using the STOMP protocol over HTML5 Web Sockets.

In "Locations Application Using STOMP" on page 6, we described the Locations application. In this chapter, we will write the web application (shown in Figure 3-1) that consumes the device's GPS position to display it on a map and send text messages to the device.

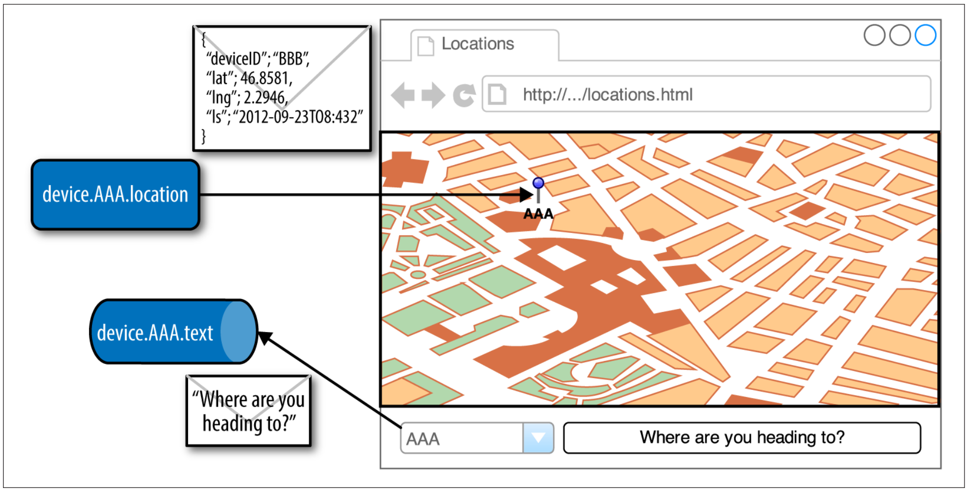

{
"deviceID"; "BBB",
"lat"; 46.8581,
"lng"; 2.2946,
"ls"; "2012-09-23T08:432"
}
device.AAA.location
device.AAA.text
"Where are you heading to?"
Locations
http://.../locations.html
AAA
AAA
Where are you heading to?

Figure 3-1. Diagram of the Locations web application

Throughout the chapter, we will show all the code required to run the application.

The whole application code can be retrieved from the GitHub repository in the *stomp/web/* directory.

HTML5 Web Sockets

Web Sockets is a recent protocol that provides a bidirectional and full-duplex communication channel over a single TCP connection between the web browser and the web server. Before Web Sockets, communication was only one way: the browser initiated communication by sending an HTTP request to the web server, which replied with an HTTP response. The server could not send data to the browser without the browser having first sent a request. Some techniques exist to provide a bidirectional HTTP (such as HTTP long polling and HTTP streaming), but they have significant issues, as described in RFC-6202.

Thanks to HTML5 Web Sockets, it is now possible to have two-way communication between the browser and the server. The server can send data to the browser without waiting for a request. This enables us to deliver messages from the web server to the browser.

We are no longer limited to pull-based messages where the browser needs to regularly contact the server to know whether there are any available messages to consume. This preserves bandwidth (the browser no longer needs to send periodic query requests) and battery on mobile devices.

Web Sockets have many use cases in addition to messaging. In some simple cases, sending data on Web Sockets without using a messaging protocol could be enough.

For further information, the book *High Performance Browser Networking* contains a thorough introduction to Web Sockets.

stomp.js, STOMP over Web Sockets

If your web browser supports Web Sockets (and all modern browsers do), we can then use a STOMP client that leverages them to send and receive STOMP messages directly from it.

stomp.js is a small JavaScript library (8 kibibytes for its minified version) that provides a full-featured STOMP 1.2 client.

It can run in multiple JavaScript platforms. Its main target is the web browser (using Web Sockets as its transport protocol), but it can also be used on other JavaScript platforms such as node.js (using TCP sockets).

A STOMP client running in a web browser is similar to another STOMP client running on another platform with one exception: STOMP messages are transported by Web Sockets and not regular TCP sockets. The significant difference between them is that a Web Socket is created by sending an HTTP request with an `Upgrade: websocket` header. The STOMP broker must be able to handle such an HTTP request and upgrade the connection from HTTP to the STOMP protocol.

Web Sockets support is not part of the STOMP protocol. The most commonly used STOMP brokers—such as ActiveMQ (which is used in this book), HornetQ, and RabbitMQ—support it, but you will have to consult your broker documentation to check if it does.

Bootstrap the Locations Web Application

As we explained in the first chapter, this web application will be used to display the locations of the devices and send them text messages.

It will be a very simple one-page web application that can be run from a web server serving static pages. It does not require any server-side runtime, as all the code will be executed inside the web browser using JavaScript.

As shown in Example 3-1, let's bootstrap the web application by creating a *locations.html* page.

Example 3-1. Template for the locations.html web application

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta content="width=device-width" name="viewport">
    <meta charset="utf-8">
    <title>Locations - STOMP Example</title>
  </head>
  <body>
    <h1>Locations - STOMP Example</h1>
    <form id='connect_form'>
      <fieldset>
        <label>WebSocket URL</label>
        <input name=url id='connect_url'
          value='ws://localhost:61614' type="text">
        <button id='connect_button' type="submit">Connect</button>
        <button type="button"
          id='disconnect_button' disabled>Disconnect</button>
      </fieldset>
    </form>
    <form id="text_form" style="display: none;">
      <fieldset>
```

```
        <legend>Send Text</legend>
        Device: <select id="deviceID"></select>
        <br>
        Text: <input id='text' size=64
          placeholder="type the text to sent to the device" type="text">
        <br>
        <button id='text_submit' type="submit">Send</button>
      </fieldset>
    </form>
    <hr>

    <div id="map-canvas" style="height:512px; width:100%; padding:0; margin:0">
    </div>

    <footer>&copy; 2014 <a href="http://mobile-web-messaging.net">Mobile &amp;
      Web Messaging</a></footer>

    <!-- Scripts placed at the end of the document so the pages load faster -->
     <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js"></
script>
       <script  src="https://maps.googleapis.com/maps/api/js?v=3.exp&sensor=false"></
script>
      <script  src="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.1/stomp.js"></
script>
    <script>
$(document).ready(function() {

// We will put all the JavaScript code in this is called
// when the document is ready

};
    </script>
  </body>
</html>
```

Create a Stomp Client with stomp.js

The STOMP client will be used both for consuming messages (from the device's location destination) and producing them (to the device's text destination).

We will declare the `client` variable to use it throughout our script:

```
$(document).ready(function() {

  var client; // keep a reference on the Stomp client that we create

}
```

When the user clicks on the `Connect` button, the URL is retrieved from the `connect_url` input element and passed to the `connect` method. In this method, we will create the `client` and connect to the STOMP broker:

```
// handles the connect_form
$('#connect_form').submit(function() {
  var url = $("#connect_url").val();

  connect(url);
  return false;
});
```

Connect to a STOMP Broker

In the `connect` method, we create the STOMP client by calling `Stomp.client(url)`. When we have the `client` object, we call its `connect` method to connect to the STOMP broker. This method takes at least two arguments:

- A set of headers containing additional metadata that the STOMP broker can use during the connection process (in our case, we have no such header, so we will pass an empty JavaScript literal object `{}`)
- A `connectedCallback` function that is called back when the client is successfully connected to the STOMP broker)

```
// connection to the STOMP broker
// and subscription to the device's position destinations
//
// the url paramater is the Web Socket URL of the STOMP broker
function connect(url) {

  // creates the callback that is called when the client
  // is successfully connected to the STOMP broker
  var connectedCallback =  function(frame) {
      ...
  };

  // create the STOMP client
  client = Stomp.client(url);
  // and connect to the STOMP broker
  client.connect({}, connectedCallback);
}
```

The `connectedCallback` is called when the client is successfully connected to the STOMP broker. At this stage, we can clean up the user interface by hiding the `Connect` button, and showing the `Disconnect` button and the form to send text messages to the devices:

```
var connectedCallback =  function(frame) {
  client.debug("connected to Stomp");
  // disable the connect button
  $("#connect_button").prop("disabled",true);
  // enable the disconnect button
  $("#disconnect_button").prop("disabled",false);
  // show the form to send text to the devices
  $("#text_form").show();

  // ...

};
```

If we want to be notified when the connection is *unsuccessful*, we can pass an additional argument to the `connect` method: a function that is called back if the connection is *not* successful.

```
client.connect({}, connectedCallback,
  function(frame) {
    // this function is executed if the connection to the STOMP broker fails
});
```

Receive STOMP Messages

When the client is connected successfully to the STOMP broker, it can subscribe to a destination using the `subscribe` method, which takes two parameters:

- The name of the destination
- A function that is called back every time the broker delivers a message to the client:

```
client.subscribe(destination, function(message) {
  // this function is executed every time a message is consumed
});
```

The `message` parameter that is passed to the subscription callback corresponds to a STOMP message and has three properties:

`command`
: The command of the STOMP frame (when a message is received, it will always be `MESSAGE`)

`headers`
: A JavaScript object containing all the frame headers; it can be empty if the message has no headers

`body`
: A string representing the message's payload; it can be `null` if the message has no payload

Subscribe to a Wildcard Destination

This web application wants to receive the location of *any* devices that broadcast it. This means that we must subscribe to the /topic/device.XXX.location for every device where XXX is the device identifier.

There are two different ways to achieve this. The first way is to know beforehand all the device IDs and subscribe to their topics one after the other. We can use the same subscription callback for all of them. However, that implies that the web application must now have a way to know this list. For example, it could be a web service that returns such a list.

The pseudocode for it would look like:

```
var listURL = "...";
var deviceIDs = fetch(listURL);
var callback = function(message) {
  // we use the same callback for every subscription
}
for (deviceID in deviceIDs) {
  var destination = "/topic/truck." + deviceID + ".location";
  client.subscribe(destination, callback);
}
```

But what happens if another device starts broadcasting its location *after* the web application fetches the list of device IDs? The web application will not subscribe to its topic and will never display it on the map. We would have to periodically fetch the list of device IDs and check whether there are new ones or if some devices have been removed.

Fortunately, the flexibility of the STOMP protocol comes in handy to manage this in a simpler fashion. The STOMP 1.2 protocol defines very loosely the notion of destination:

> *A STOMP server is modelled as a set of destinations to which messages can be sent. The STOMP protocol treats destinations as opaque string and their syntax is server implementation specific. Additionally STOMP does not define what the delivery semantics of destinations should be. The delivery, or "message exchange", semantics of destinations can vary from server to server and even from destination to destination. This allows servers to be creative with the semantics that they can support with STOMP.*

Until now, we have used *simple* destinations (e.g., /topic/device.2262EC25-E9FD-4578-BADE-4E113DE45934.location or /queue/device.2262EC25-E9FD-4578-BADE-4E113DE45934.text) that are straightforward to understand.

We will now use a feature from our STOMP broker, ActiveMQ, that allows us to use *wildcard* destinations.

- . is used to separate names in a path

- `*` is used to match any name in a path
- `>` is used to recursively match any destination starting from this name

With our example using ActiveMQ, we can use this notation to subscribe to any device location topic by using the `/topic/device.*.location` wildcard destination (where `*` stands for *any device identifier*).

The subscription code becomes simpler:

```
// we use a wildcard destination to register to any
// destination that matches this pattern.
var destination = "/topic/device.*.location";
client.subscribe(destination, function(message) {
  // this function is called every time a message is received
});
```

The semantics of STOMP destinations are specific to the STOMP broker so you have to check its documentation to determine if wildcard destinations or similar concepts are supported. If it does not, you have to revert to the first idea to fetch the list of devices and subscribe to each of the destinations, or use another STOMP broker that supports this feature.

Because we no longer know *a priori* which device location we are receiving, we need a new way to determine which device broadcast it. There are two pieces of information we can use. When a consumer receives a STOMP message, the message always has a `destination` header that corresponds to the *actual* destination from which the client consumed.

We are subscribing to the wildcard address `/topic/device.*.location`, so the actual destination from which we consume will be stored in the received message's headers in `message.headers["destination"]`. In my case, the value of this header would be `/topic/device.2262EC25-E9FD-4578-BADE-4E113DE45934.location`. However, we would then have to parse this `destination` to extract the device ID from it and write brittle code for that.

If we look back at "Message representation for the Locations application" on page 7, the message representation for the location also contains the device ID in the `deviceID` property:

```
{
  "deviceID": "BBB",
  "lat": 48.8581,
  "lng": 2.2946,
  "ts": "2013-09-23T08:43Z"
}
```

The message is *self-contained* and defines all the interesting information that a consumer might need. When we receive such a location message, we know which device is sending it by simply looking at the `deviceID` property from the JSON object created from the message body:

```
var destination = "/topic/device.*.location";
client.subscribe(destination, function(message) {
  // this function is called every time a message is received
  // create an object from the JSON string contained in the message body
  var payload = JSON.parse(message.body);

  var deviceID = payload.deviceID;

  //...
});
```

When we receive the position of a device, the last step we need to make is to display its position on a map. We will wrap this code in a `show` method that is called from the subscription callback. The callback will pass three parameters to this method: the device identifier, its latitude, and its longitude.

The code to connect to the STOMP broker and subscribe to the wildcard destination is shown here:

```
// connection to the STOMP broker
// and subscription to the device's position destinations
//
// the url paramater is the Web Socket URL of the STOMP broker
function connect(url) {

  // creates the callback that is called when the client
  // is successfully connected to the STOMP broker
  var connectedCallback =  function(frame) {
    client.debug("connected to Stomp");
    // disable the connect button
    $("#connect_button").prop("disabled",true);
    // enable the disconnect button
    $("#disconnect_button").prop("disabled",false);
    // show the form to send text to the devices
    $("#text_form").show();

    // after the client is connected, subscribe to
    // the device's location destinations

    // we use a wildcard destination to register to any
    // destination that matches this pattern
    var destination = "/topic/device.*.location";
    client.subscribe(destination, function(message) {
      // this function is called every time a message is received
      // create an object from the JSON string contained in the message body
      var payload = JSON.parse(message.body);
      var deviceID = payload.deviceID;
      if (!$("#deviceID option[value='" + deviceID + "']").length) {
        // if the device ID is not already in the list of devices we can send
        // orders to, we add it.
        $('#deviceID').append($('<option>', {value:deviceID}).text(deviceID));
      }
      // show the device location on the map
      show(deviceID, payload.lat, payload.lng);
    });
  };

  // create the STOMP client
  client = Stomp.client(url);
  // and connect to the STOMP broker
  client.connect({}, connectedCallback);
}
```

Draw the Device Locations on a Map

The web application is now receiving the GPS coordinates of any devices that send them. We could just display them as text like we did for the iOS application in "Display the Device Position" on page 26, but we can make it prettier by drawing them on a map using the Google Maps API.

In the template in Example 3-1, we already added the scripts to use Google Maps API. We now need to create the map and initialize it:

```
$(document).ready(function() {

  // Google map and the trackers to follow the trucks
  var map, trackers = {};

  function initialize() {
    var mapOptions = {
      zoom: 2,
      center: new google.maps.LatLng(30,0),
      mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    map = new google.maps.Map($("#map-canvas").get(0), mapOptions);
  }

  // initialize the Google map
  google.maps.event.addDomListener(window, 'load', initialize);
```

With this initialization code, the map will be drawn in the `map_canvas` `div` element and we can reference it using the `map` variable.

The `trackers` variable is a map whose keys are the device identifiers and the values are a tracker with the latest location of the device on the map.

We called a show() method in the subscription handler. We can now implement it to display the device on the map using its coordinates:

```
// show the device at the given latitude and longitude
function show(deviceID, lat, lng) {
  var position = new google.maps.LatLng(lat, lng);
  // lazy instantiation of the map
  if (!map) {
    create_map(position);
  }
  if (trackers[deviceID]) {
    // the tracker is known; we just need to update its position
    trackers[deviceID].marker.setPosition(position);
  } else {
    // there is no tracker for this device yet; let's create it
    var marker = new google.maps.Marker({
      position: position,
      map: map,
      title: deviceID + " is here"});
    var infowindow = new google.maps.InfoWindow({
      content: "Device " + deviceID
    });
    var tracker = {
      marker: marker
    };
    // add it to the trackers
    trackers[deviceID] = tracker;
    google.maps.event.addListener(marker, 'click', function() {
      infowindow.open(map, marker);
    });
  }
}
```

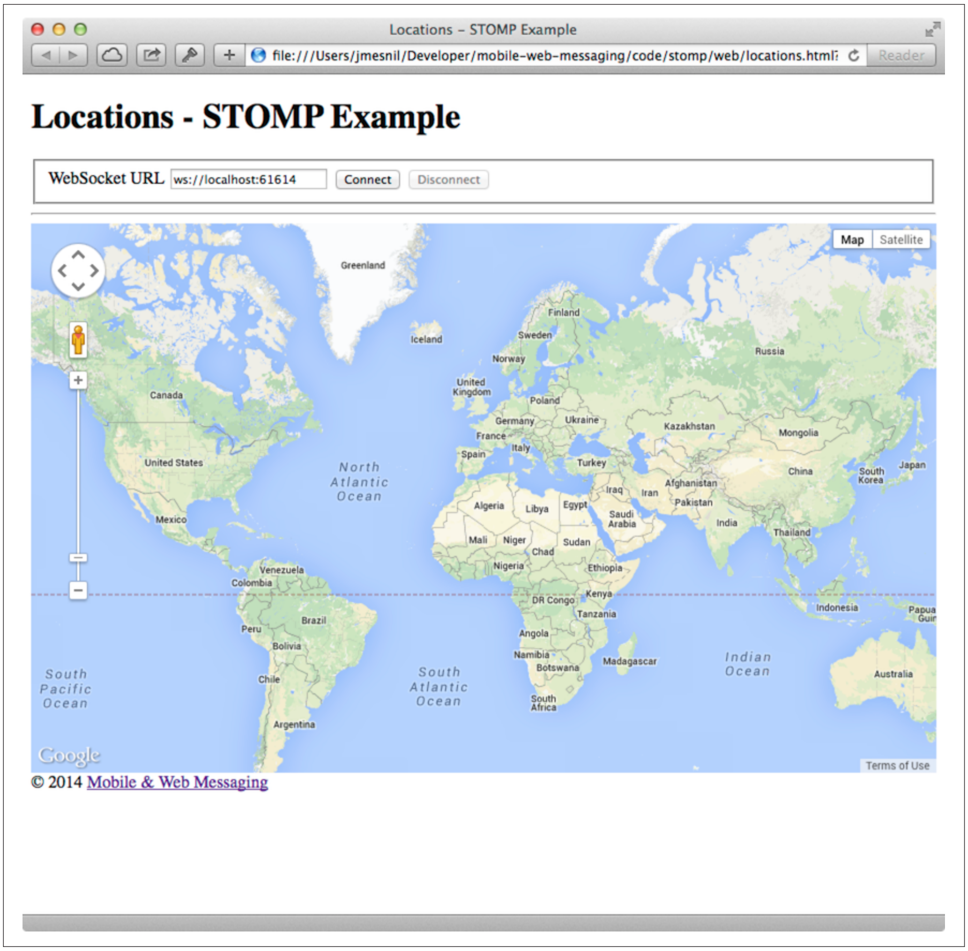
If we now open this *locations.html* file in a web browser, we will see a map of the whole world displayed (Figure 3-2).

If we click on the Connect button, markers will appear on the map for each device that broadcasts its coordinates.

In my case, I am using the iOS simulator to run the mobile application developed in . I use the Location tool to simulate a freeway drive (as explained in "Simulate a Location with iOS Simulator" on page 32). See Figure 3-3.

The position of the device is updated every time the web application receives a STOMP message from the device's position destination and we see it move on the map.

At this stage, the web application receives STOMP messages to display the position of the devices. We now need to write the code to send texts to the devices.


Locations – STOMP Example
file:///Users/jmesnil/Developer/mobile-web-messaging/code/stomp/web/locations.html
Reader
Map
Satellite
Google
Terms of Use

Locations - STOMP Example

WebSocket URL ws://localhost:61614 Connect Disconnect

© 2014 Mobile & Web Messaging

Figure 3-2. The Locations web application

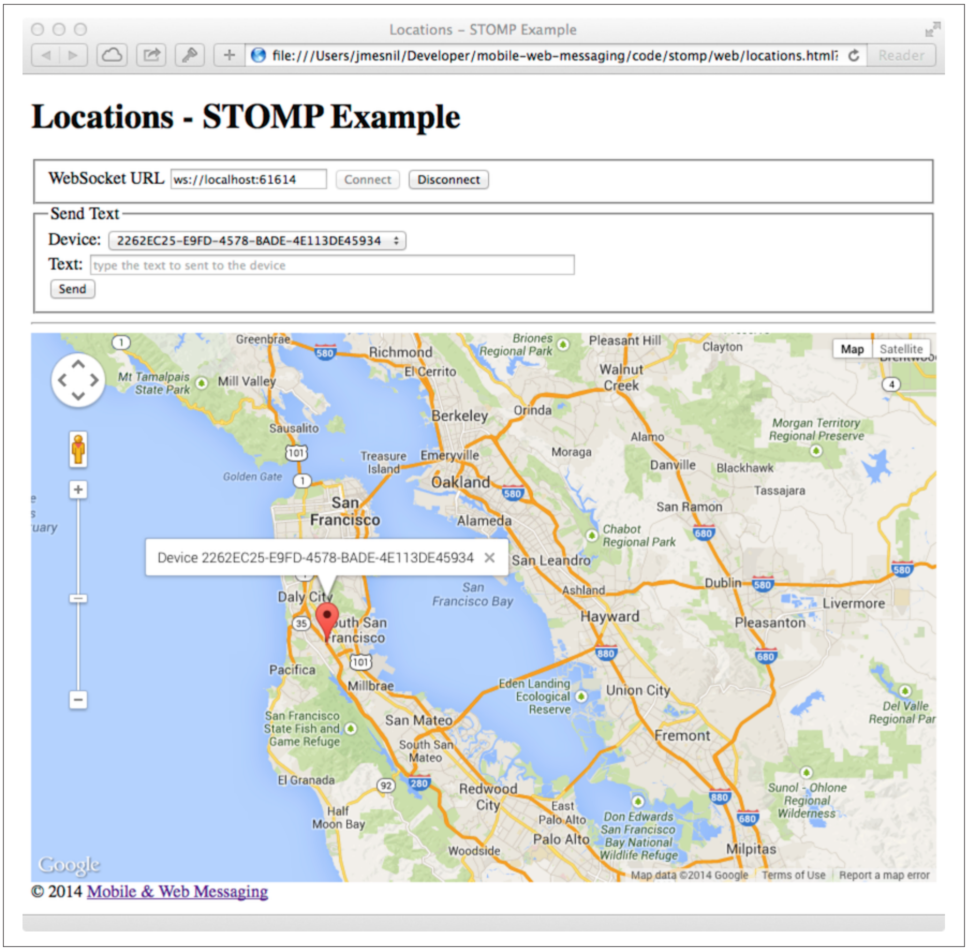

Locations – STOMP Example
file:///Users/jmesnil/Developer/mobile-web-messaging/code/stomp/web/locations.html
Reader
Locations - STOMP Example
WebSocket URL ws://localhost:61614
Connect
Disconnect
Send Text
Device: 2262EC25-E9FD-4578-BADE-4E113DE45934
Text: type the text to sent to the device
Send
Device 2262EC25-E9FD-4578-BADE-4E113DE45934
Map
Satellite
San Francisco
Oakland
Berkeley
Richmond
Hayward
Fremont
San Mateo
Palo Alto
Pleasanton
Livermore
© 2014 Mobile & Web Messaging

Figure 3-3. Simulating a freeway drive

Send STOMP Messages

The STOMP client can send messages to the broker using its `send` method, which takes three parameters:

`destination`
: The name of the destination

`headers`
: A JavaScript object containing any additional headers

`body`
: A string corresponding to the message payload

Both headers and body are optional and can be omitted. However, if you want to set the message body, you must also specify the headers (using an empty JavaScript literal if you have no header to set):

```
client.send(destination, {}, body);
```

As we described in "Messaging models for the Locations application" on page 6, we use a queue to send orders to a given device, and the destination for this is named /queue/device.XXX.text.

The text is sent in the STOMP message body as a plain-text string:

```
Hello, where are you?
```

We must respect this message format, as it is the format expected by the iOS application to handle the texts and display them (we wrote this code in "Subscribe to a STOMP Destination" on page 49).

We added an HTML <form> element with the id text_form to send a text message. The device identifier is taken from the selected option in the <select> element identified by deviceID. The text itself is taken from the <input> element identified by text.

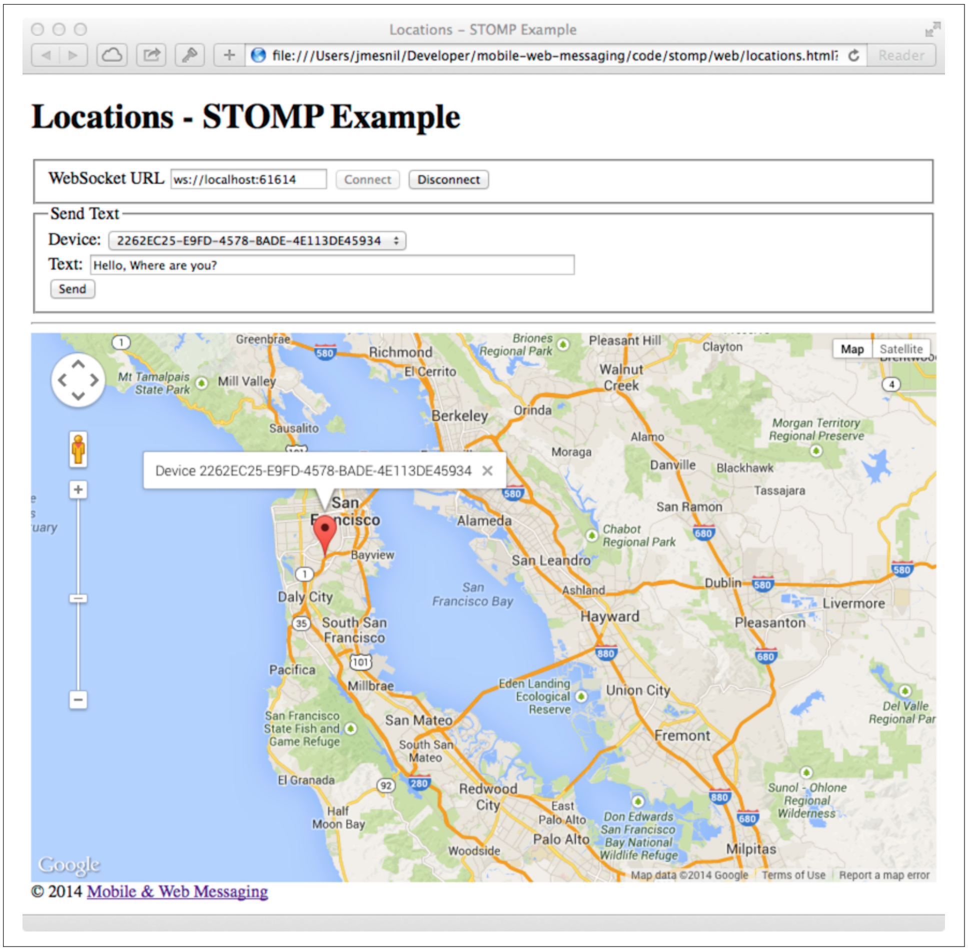
When we know the deviceID and the order, we have all we need to send an order to this device. The destination for the order will be built using the device.

Piecing everything together, the code to send a STOMP message looks like:

```
// send a text to a device
$('#text_form').submit(function() {
  var deviceID = $("#deviceID").val();
  var text = $("#text").val();

  // use the device's queue orders as the destination
  var destination = "/queue/device." + deviceID + ".text";
  // text is sent as a plain-text string
  client.send(destination, {}, text);
  // reset the text input field
  $("#text").val("");
  return false;
});
```

If we reload the *locations.html* file after adding this code, we can now send any text message to a device by selecting it in the list in the Send Text form.

As illustrated in , let's type "Hello, where are you?" and click on the Send button.


Locations – STOMP Example

file:///Users/jmesnil/Developer/mobile-web-messaging/code/stomp/web/locations.html

Locations - STOMP Example

WebSocket URL: ws://localhost:61614 Connect Disconnect

Send Text

Device: 2262EC25-E9FD-4578-BADE-4E113DE45934

Text: Hello, Where are you?

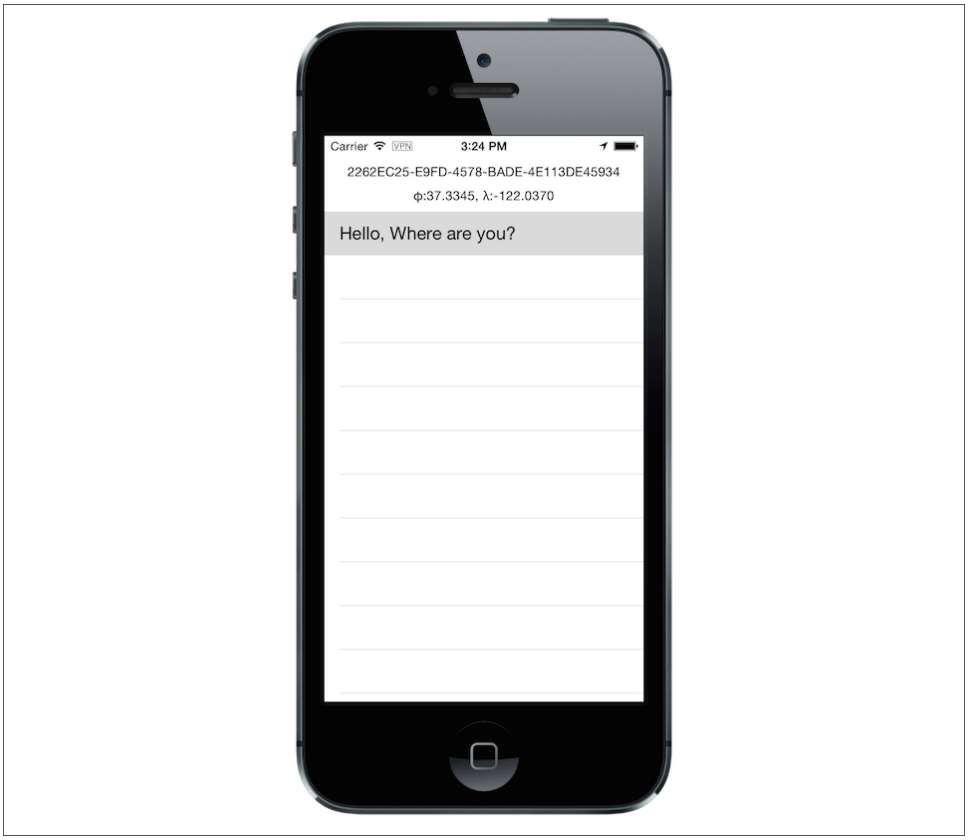
Send

Device 2262EC25-E9FD-4578-BADE-4E113DE45934

© 2014 Mobile & Web Messaging

Figure 3-4. Send a text message to a device

The message is sent when you click on the Send button. The Locations iOS application is subscribed to this destination, so it will receive the message and display it in its table (Figure 3-5).



2262EC25-E9FD-4578-BADE-4E113DE45934

ϕ:37.3345, λ:-122.0370

Hello, Where are you?

Figure 3-5. The Locations iOS application received the text

Disconnect from the STOMP Broker

locations.html connects to the STOMP broker when the user clicks on the Connect button. The user can then disconnect from the broker by clicking on the Disconnect button.

To disconnect from the STOMP broker, we call the `disconnect` method on the `client` and pass a callback that is called after the disconnection succeeds.

In this callback, we clean up the trackers that are displayed on the map so that nothing is shown after the user is disconnected. We also revert the user interface by hiding the Disconnect button and the text form and showing the Connect button:

```
function disconnect() {
  // disconnect from the broker
  client.disconnect(function() {
    // when we are successfully disconnected, clear out all the trackers from
    // the map
```

```
      for (var tracker in trackers) {
        trackers[tracker].marker.setMap(null);
      }
      trackers = {};
    });
    $("#deviceID").empty();
    $("#connect_button").prop("disabled",false);
    $("#disconnect_button").prop("disabled",true);
    $("#text_form").hide();
  }
```

Finally, we need to call this disconnect method when the `Disconnect` button is clicked:

```
  $('#disconnect_button').click(function() {
    disconnect();
    return false;
  });
```

Summary

In this chapter, we learned to use stomp.js to send and receive STOMP messages from a web application.

Whether you are using StompKit in an iOS application or stomp.js in a web application, the steps are always the same.

To send a message, the application must do the following:

1. Connect to the STOMP broker.
2. Send the message to the destination.

To consume a message, the application must do the following:

1. Connect to the STOMP broker.
2. Subscribe to the destination and pass a callback that is called every time a message is received.

At the end of this chapter, we have a very simple application that works. If you have access to several iPhone devices, you can see that the web application will display the location of all the devices running the iOS application.

In the next chapter, we will learn about more advanced features of STOMP. We did not present them earlier because they were not required to write this simple application. However, it is likely that you may need some of these features if your applications are more complex than this simple example.

CHAPTER 4

Advanced STOMP

In Chapters 2 and 3, we used STOMP to send and receive messages from a native iOS application and a web application. STOMP provides additional features that we did not use to write these applications. In this chapter, we will take a tour of all the features provided by STOMP.

This chapter covers the latest version of the protocol at the time of this writing (i.e., STOMP 1.2, which was released on October 22, 2012).

As we present these features, we will show how to use them from either StompKit or stomp.js. The API may vary depending on the platform and language, but the concept remains the same. If you ever need to use STOMP from another language or platform, you will have to adapt to the library API—but the underlying concept will still apply.

Clients and brokers can use these features by sending additional frames (defined in the protocol) or by using message headers. The STOMP protocol defines frames to provide transactions, a combination of frames and headers for message acknowledgment, and headers for heartbeat negotiation.

In this chapter, we will describe the features provided by the STOMP protocol that the client and broker must support in order to be compliant.

STOMP is also extensible, and clients and brokers can support additional features by using headers not defined in the protocol. We will describe these additional features in .

In any case, the features (defined in the protocol or supported only by some brokers) rely on the structure of STOMP frames.

Frame Representation

All data exchanged between a client and a broker is done using STOMP frames. STOMP is modeled on HTTP and its frames follow a similar structure. Each frame is composed of three elements (Example 4-1):

- A *command*
- A set of *headers*
- A *payload* (optional)

Example 4-1. STOMP Frame Structure

```
COMMAND ❶
header1:value1 ❷
header2:value2
❸
payload^@ ❹
```

❶ A frame starts with a command string followed by an end-of-line (EOL)

❷ Header entries follow the format `<key>:<value>` and are ended by EOL

❸ A blank line separates the set of headers from the payload

❹ A frame is ended by a NULL octet (represented by `^@` in ASCII)

STOMP is based on text (using UTF-8 for its default encoding), but it can also transmit binary data in its payload by specifying an alternative encoding.

In Chapters 2 and 3, we were sending JSON data in the messages. We were setting the string representation of the JSON structure in the message body and setting the `content-type` header to `application/json; charset=utf-8`.

STOMP uses frames not only to send messages (with the `SEND` command) and deliver them (with the `MESSAGE` command) but also for all its operation commands (such as `CONNECT`, `DISCONNECT`, `SUBSCRIBE`, etc.).

Headers

STOMP defines only a handful of headers for its different frames.

It is also possible to add other headers when sending a frame, as long as it does not collide with the headers already specified in the STOMP protocol.

An application can use headers instead of the message's payload to transmit data. The fundamental difference between headers and payload is that headers can be read (and modified) by the STOMP brokers. Payload is treated as a black box and the broker never reads or modifies it.

In practice, headers are most often used to specify additional features or constraints to the message processing.

Authentication

In Chapters 2 and 3, the STOMP clients are connected to the broker *anonymously*. We do not pass any user credentials that the broker could use to authenticate the user and check whether he or she can send and/or receive messages.

If the STOMP broker is configured to accept secured connections, the client needs to pass `login` and `passcode` parameters when it connects to the STOMP broker.

By default, ActiveMQ accepts anonymous connections. To change its security settings to authenticate users and grant them restricted privileges (so that they can only receive messages from example), consult its security documentation page.

StompKit Example

To use an authenticated connection with `StompKit`, you need to pass the login and passcode parameters to the headers dictionary when calling the `connectWithHeaders:completionHandler` method on the `STOMPClient`. `StompKit` defines the `kHeaderLogin` and `kHeaderPasscode` constants to represent the headers keys:

```
- (void)connect
{
    NSLog(@"Connecting...");

    NSString *login = @"...";
    NSString *passcode = @"...";
    [self.client connectWithHeaders:@{ @"client-id": self.deviceID,
                                       kHeaderLogin: login,
                                       kHeaderPasscode: passcode }
                  completionHandler:^(STOMPFrame *connectedFrame,
                                      NSError *error) {
                      ...
                  }];
}
```

stomp.js Example

A stomp.js client can be authenticated by adding the `login` and `passcode` headers to the first parameter of its `connect` method:

```
function connect(url) {
  // create the STOMP client
  client = Stomp.client(url);

  ...

  var connectedCallback =  function(frame) {
    ...
  };

  var login = ...;
  var passcode = ...;

  client.connect({
      login: login,
      passcode: passcode,
    }, connectedCallback);
}
```

Message Acknowledgment

Message acknowledgment is a feature available to STOMP *consumers*.

When the broker delivers a message to a consumer, there is a transfer of responsibility between the broker and the consumer to determine which is the *owner* of the message. The consumer becomes responsible for the message by *acknowledging* the message.

By default, the STOMP broker will consider the message automatically acknowledged when it is delivered to the consumer.

However, there are cases in which the consumer may prefer to explicitly acknowledge the message. It leaves a window of opportunity to determine whether it can handle the message or not. For example, the client needs to write the message payload in a data store. There may be issues with opening a connection to the data store and the client could choose to acknowledge the message only after having successfully written its body to the data store. In case of failure, it will instead *nack* the message (explicitly refuse to take ownership of it). When the STOMP broker is informed of this negative acknowledgment, it may then decide to deliver the message to another consumer subscribed to the destination or try again some time later depending on its configuration.

The consumer specifies its type of acknowledgement when it subscribes to a destination. STOMP supports three types of acknowledgements:

- `auto` (by default)
- `client`

- `client-individual`

If the client does not specify any type of acknowledgment or if it uses `default`, it does not need to send any acknowledgment; the STOMP broker will consider the message acknowledged as soon as it is delivered to the client.

If `client` or `client-individual` is used, the consumer must send acknowledgments to the server with the `message-id` of the message that is acknowledged. The difference between `client` and `client-individual` is that `client` will acknowledge the message *and all other messages previously delivered to the consumer* `client-individual` will only acknowledge the message and no other messages. The consumer acknowledges a message by sending an `ACK` frame to the STOMP broker.

If `client` and `client-individual` are used, the consumer can explicitly refuse to handle the message by sending a `NACK` frame, which is a negative acknowledgment.

StompKit Example

The message acknowledgment is specified when the `STOMPClient` subscribes to a destination by calling its `subscribeTo:headers:messageHandler:` method. To specify a `client` or `client-individual` acknowledgment, you must set an `ack` header. *StompKit.h* defines constants to represent the header name, `kHeaderAck`, and its accepted values, `kAckAuto`, `kAckClient`, and `kAckClientIndividual`.

The `STOMPMessage` parameter of the `messageHandler` has two methods, `ack` and `nack`, to acknowledge or nack the message, respectively.

If the `ack` header is not set or if it set to `auto`, message acknowledgment is performed by the broker and calling the `STOMPMessage`'s `ack` and `nack` methods will do nothing:

```
// use client acknowledgment
[self.client subscribeTo:destination
                 headers:@{kHeaderAck: kAckClient}
          messageHandler:^(STOMPMessage *message) {
              // process the message
              // ...

              // acknowledge it
              [message ack];
              // or nack it with
              // [message nack]
          }];
```

stomp.js Example

The `client` can specify the type of acknowledgment by passing a dictionary with the `ack` header as the last parameter of its `subscribe` message.

The message parameter of the subscribe callback has two methods, ack and nack, to acknowledge or nack the message, respectively. If the acknowledgment type is auto (or if it is not specified at all), these ack and nack methods will do nothing:

```
client.subscribe(destination,
  function(message) {
    // process the message
    ...

    // acknowledge it
    message.ack();
    // or you can nack it by calling message.nack() instead.
  },
  {"ack": "client"}
);
```

There are many use cases where it is not necessary to use explicit acknowledgment.

For example, in the Locations web application, we do not need to acknowledge every message that we receive from the devices with its GPS position. At worst, there may be a problem displaying the position, but we know that other messages will come later to update the device's position.

Besides, acknowledging every message would have a performance cost. Sending the acknowledgment back to the broker would involve an additional network trip for every message.

The Locations iOS application is also consuming messages from the device's text queue. These messages may be more important to acknowledge explicitly. We could enhance the application by letting the user confirm that it has read the message's text and the message would be acknowledged after this confirmation only.

We could also let the user reject it by negatively acknowledging the message. In that case, these *nacked* messages would be handled back by the STOMP broker. Depending on the broker you use, it may provide additional features to handle these messages. Messages that are nacked multiple times from a destination are commonly sent to a dead letter queue. An administrator can then inspect this dead letter queue to determine what to do with these messages. For example, it can send them to another device, send alerts about the device that rejected them, and so on.

Transactions

STOMP has basic support for transactions.

Sending a message or acknowledging the consumption of messages can be performed inside a transaction. This means that the messages and acknowledgments are not processed by the broker when it receives the corresponding frames but when the transaction completes. If the client does not complete the transaction or aborts it, the broker will not process the frames that it received inside the transaction and will just discard them. Transactions ensure that messages and acknowledgment processing are *atomic*. *All* transacted messages and acknowledgments will be processed by the broker when the transaction is committed or *none* will be if the transaction is aborted.

A transaction is started by the client by sending a `BEGIN` frame to the broker. This frame must have a `transaction` header whose value is a transaction identifier that must be unique within a STOMP connection.

Sending a message can then be part of this transaction by adding a `transaction` header to its `SEND` frames set to the same transaction identifier. If a consumer is subscribed to a STOMP destination with `client` or `client-individual` acknowledgment modes, it can also make the message acknowledgment (or nack) inside a transaction by setting the `transaction` header on the `ACK` (or `NACK`) frame.

By default, STOMP consumers use `auto` acknowledgment. In that case, the message acknowledgment is performed automatically by the STOMP broker when the message is delivered to the client and the acknowledgment *cannot* be put inside a transaction.

To complete this active transaction and allow the broker to process it, the client must send a `COMMIT` frame with the same `transaction` header as in the corresponding `BEGIN` frame that started the transaction. To abort (or roll back) a transaction and discard any messages or acknowledgments inside it, the client must instead send an `ABORT` frame with this `transaction` header.

Beginning a transaction is not sufficient to send subsequent messages inside it. If a transaction is begun, the message to send must have its `transaction` header set to the transaction identifier. Otherwise, the STOMP broker will not consider the message to be part of the transaction and will process it when it receives it instead of waiting for the transaction to complete. If the client decides to abort the transaction, the message will have already been processed by the broker and will not be discarded.

STOMP does not provide a transaction timeout that would abort the transaction if it is not completed in a timely fashion. The transaction lifecycle (controlled by `BEGIN` and `COMMIT`/`ABORT` frames) is the responsibility of the client. However, the broker will automatically abort any active transaction if the client sends a `DISCONNECT` frame or if the underlying TCP connection fails.

StompKit Example

The `STOMPClient` can begin a transaction by calling its `begin:` method and passing an `NSString` that will be used to identify the transaction. Alternatively, you can call its `begin` method (without any parameter) and a transaction identifier will be automatically generated. Both `begin:` and `begin` methods return a `STOMPTransaction` object. This object has a `identifier` property that contains the transaction identifier.

Sending, acknowledging, or nacking a message can then be part of a transaction by adding a `transaction` header set to the transaction identifier (StompKit.h defines a `kHeaderTransaction` to represent this `transaction` header).

Finally, the `STOMPTransaction` object has two methods, `commit` and `abort`, to commit or rollback the transaction, respectively:

```
// begin a transaction
STOMPTransaction *transaction = [self.client begin];
// or STOMPTransaction *transaction = [self.client begin:mytxid];
NSLog(@"started transaction %@", transaction.identifier);

// send a message inside a transaction
[self.client sendTo:destination
            headers:@{kHeaderTransaction: transaction.identifier}
               body:body];

// acknowledge a message inside a transaction
[message ack:@{kHeaderTransaction: transaction.identifier}];
// or nack a message inside a transaction with
// [message nack:@{kHeaderTransaction: transaction.identifier}];

// commit the transaction
[transaction commit];
// or abort it
[transaction abort];
```

stomp.js Example

The API is very similar to StompKit. The `client` object has a `begin` method that can take a parameter corresponding to the transaction identifier. If there is no parameter, an identifier is automatically generated. The `begin` method returns a `transaction` object that has an `id` property corresponding to the transaction identifier.

Sending, acknowledging, or nacking a message can be part of a transaction by passing a `transaction` header set to the transaction identifier to these methods.

Finally, committing or aborting a transaction is performed by calling the `commit` or `abort` method, respectively, on the `transaction` object:

```
// begin a transaction
var tx = client.begin();
// or var tx = client.begin(mytxid);
console.log("started transaction " + tx.id);

// send a message inside a transaction
client.send(destination, {transaction: tx.id}, body);

// acknowledge a message inside a transaction
var subscription = client.subscribe(destination,
    function(message) {
      // do something with the message
      ...
      // and acknowledge it inside the transaction
      message.ack({ transaction: tx.id});
      // or nack it inside the transaction
      // message.nack({ transaction: tx.id});
    },
    {ack: 'client'}
  );

// commit the transaction
tx.commit();
// or abort it
tx.abort();
```

Error Handling

Until now, we have used STOMP in a perfect world where no unexpected problems happened. Realistically, problems will occur. On mobile devices, the network will be lost and the connection to the STOMP broker will be broken.

STOMP provides basic support to handle errors. The STOMP broker can inform the client that an error occurs by sending an `ERROR` frame to the client. This frame can contain a `message` header that contains a short description of the error. Most STOMP brokers deliver `ERROR` frames with a message payload containing more detailed information on the error.

STOMP specifies that after delivering an `ERROR` frame to the client, the broker must close the connection. This means that STOMP is not resilient to error. If a single error occurs on the server, the broker will close the connection to the client. In addition, there is no guarantee that the client will be able to receive the `ERROR` frame before the connection is closed.

In practice, this implies that to be able to handle any errors in the client, we should:

1. Handle `ERRORS` frames coming from the broker

2. Handle unexpected connection closed events

StompKit Example

We will modify the Locations iOS application to handle errors and automatically reconnect to the STOMP broker after a delay.

The STOMPClient has an errorHandler property that is called if the client encounters any error. Errors can come from the STOMP protocol (when the broker delivers an ERROR frame) or from the underlying network connection (e.g., if the network is lost or if the broker closes the connection before any ERROR frame is delivered).

The errorHandler property is a block with a standard NSError parameter. If the error is coming from the STOMP broker, the corresponding STOMPFrame is stored in the error's userInfo dictionary with the key frame.

There are two places where we must handle reconnection:

- During the initial connection (e.g., if the broker is not up during the initial reconnect, we will continue to attempt to connect to it until it is up again).
- When we receive an error from the STOMPClient's errorHandler property.

In both cases, we will attempt to reconnect by disconnecting first (in the eventual case where the client is already connected), waiting for 10 seconds, and connecting again. This code can be encapsulated in a reconnect: method of the MWMViewController implementation:

```
#pragma mark - Messaging

- (void)reconnect:(NSError *)error {
    NSLog(@"got error %@", error);
    STOMPFrame *frame = error.userInfo[@"frame"];
    if (frame) {
        NSString *message = frame.headers[@"message"];
        NSLog(@"error from the STOMP protocol: %@", message);
    }
    [self disconnect];
    sleep(10);
    NSLog(@"Reconnecting...");
    [self connect];
}
```

We then must call this reconnect: method from the client's errorHandler property and the completionHandler block of its connectWithHeaders:completionHandler: method (both called from the MWMViewController connect method):

```
- (void)connect
{
    NSLog(@"Connecting...");
    __weak typeof(self) weakSelf = self;
    self.client.errorHandler = ^(NSError* error) {
        [weakSelf reconnect:error];
    };
    [self.client connectWithHeaders:@{ @"client-id": self.deviceID }
                  completionHandler:^(STOMPFrame *connectedFrame,
                                      NSError *error) {
                      if (error) {
                          NSLog(@"Error during connection: %@", error);
                          [weakSelf reconnect:error];
                      } else {
                          // we are connected to the STOMP broker without
                          // an error
                          NSLog(@"Connected");
                          [self subscribe];
                      }
                  }];
    // when the method returns, we can not assume that the client is connected
}
```

To avoid a retain/release cycle between `self` and the blocks, we need to create a *weak* reference to `self` to use it inside the blocks.

stomp.js Example

A stomp.js `client` can specify an `errorCallback` handler as the last parameter of its `connect` method. This handler will be called whenever the client encounters an error (whether coming from the STOMP protocol or the underlying network connection).

We can modify the Locations web application to automatically reconnect when an error occurs.

We will create a `reconnect` method that disconnects the stomp.js client if it is connected and calls `connect` again with the Web socket URL:

```
function reconnect(url) {
  if (client.connected) {
    console.log("disconnecting...");
    disconnect()
  }
  console.log("reconnecting");
  connect(url);
}
```

We then need to create an `errorCallback` handler that calls this `reconnect` method and pass it as the last parameter of the `client`'s `connect` method:

```
function connect(url) {
  var connectedCallback =  function(frame) {
    ...
  };
```

```
    var errorCallback = function(error) {
      client.debug("received error: " + error);
      reconnect(url);
    };

    // create the STOMP client
    client = Stomp.client(url);
    // and connect to the STOMP broker
    client.connect({}, connectedCallback, errorCallback);
  }
```

A Web socket can be opened only once. If a problem occurs and the socket is closed, it can no longer be used. This implies that we cannot just call the `client`'s `connect` method again as its Web Socket is no longer usable. Instead, we must create a *new* `client` that will open a new Web Socket.

Whenever an error occurs (e.g., if the network connection is broken or the STOMP broker becomes temporarily unavailable), the `errorCallback` will be called and the client will try to reconnect.

Depending on your application, you may instead decide to report the error to the user and let him know that the client is no longer able to exchange messages with the broker.

Receipts

STOMP provides a basic mechanism to let a client know when the broker has received and processed its frames. This can be used with any STOMP frames. For example, a client can be notified when the broker receives a message that a producer sent (using a `SEND` frame) or when a consumer subscribes to a destination (with a `SUBSCRIBE` frame).

To use this mechanism, the frame that is sent to the broker must include a `receipt` header with any arbitrary value. After the broker has processed the frame, it will deliver a `RECEIPT` frame to the client with a `receipt-id` header corresponding to the `receipt` header in the frame that has been processed.

As an example, we can use `receipt` to confirm that a consumer has been subscribed successfully to a destination. If the broker cannot successfully create the subscription, it will send back an `ERROR` frame to the client and close the connection. In practice, this means that a successful subscription is *silent*: the client is not informed of its success. We can use receipts to have an explicit confirmation of the subscription by adding a `receipt` header when the client subscribes to a destination. The broker must then deliver a `RECEIPT` frame that will inform the client that the broker has processed its subscription successfully. If the subscription is unsuccessful, the broker will deliver an ERROR frame that has a receipt-id header corresponding to the RECEIPT's re ceipt header to correlate the error.

Another use case for receipts is to make sending a message *synchronous*. The client sends a message with a receipt header and blocks until it receives the corresponding RECEIPT frame. This adds reliability (the client is sure that the broker has processed its message) at the cost of performance (the client can do nothing until the RECEIPT frame is received).

StompKit Example

A STOMPClient has a receiptHandler property that can be set to handle receipts. The receiptHandler is a block that takes a STOMPFrame corresponding to a RECEIPT frame.

Let's add a receipt for the device text queue's subscription to the Locations iOS application.

In its subscribe method, we will build a receipt identifier for the subscription receipt and set the client's receiptHandler. In this block, we just check if the headers of the frame parameter contain a kHeaderReceiptID key with a value that matches the receipt identifier.

To receive such a receipt from the subscription, we need to add a kHeaderReceipt header to the subscribeTo:headers:messageHandler: and set it to the receipt identifier:

```
- (void)subscribe
{
    // susbscribes to the device text queue:
    NSString *destination =
        [NSString stringWithFormat:@"/queue/device.%@.text", self.deviceID];

    // build a receipt identifier
    NSString *receipt = [NSString stringWithFormat:@"%@-%@",
                          self.deviceID, destination];
    // set the client's receiptHandler to handle any receipt delivered by
    // the broker
    self.client.receiptHandler = ^(STOMPFrame *frame) {
        NSString *receiptID = [frame.headers objectForKey:kHeaderReceiptID];
        if ([receiptID isEqualToString:receipt]) {
            NSLog(@"Subscribed to %@", destination);
        }
    };
    NSLog(@"subscribing to %@", destination);
    // pass a receipt header to be informed of the subscription processing
    subscription = [self.client subscribeTo:destination
                                    headers:@{kHeaderReceipt: receipt}
```

```
                                messageHandler:^(STOMPMessage *message) {
                 ...
        }];
    }
```

If the Locations iOS application is run with this code, we see the log that confirms that the client is successfully subscribed to the destination:

```
2014-04-21  17:30:39.205  Locations[2384:3903]  Subscribing  to  /queue/device.
2262EC25-E9FD-4578-BADE-4E113DE45934.text
2014-04-21  17:30:39.208  Locations[2384:3903]  Subscribed  to  /queue/device.
2262EC25-E9FD-4578-BADE-4E113DE45934.text
```

Note that the client's `receiptHandler` will receive any receipt delivered to the client. If you expect receipts from different STOMP frames, the client will have to handle all of them from a single `receiptHandler` block.

stomp.js Example

The `stomp.js` client has an `onreceipt` handler that can be set to receive receipts. It takes a function with a single `frame` parameter corresponding to a `RECEIPT` frame.

To receive a receipt for a subscription, we just need to add a `receipt` header to the headers passed as the last parameter of the `subscribe` method:

```
var destination = "/topic/device.*.location";

var receipt = "receipt_" + destination;
client.onreceipt = function(frame) {
  var receiptID = frame.headers['receipt-id'];
  if (receipt === receiptID) {
    console.log("subscribed to " + destination);
  }
}
client.subscribe(destination, function(message) {
  ...
}, {receipt: receipt});
```

If we reload the Locations web application, the browser console will display a log when the receipt confirming the subscription is handled by the client.

All `stomp.js` method that corresponds to STOMP frames accept a `headers` parameter that can be used to receive `RECEIPT` frames from the broker.

Heart-Beating

STOMP offers a mechanism to test the healthiness of a network connection between a STOMP client and a broker using heart-beating. In the absence of messages exchanged between the STOMP client and the broker, both can send a *heartbeat* periodically to inform the other that it is alive but has no activity.

If heart-beating is enabled, this allows the client and the broker to be informed in case of network failures and act accordingly (the client could try to reconnect to the broker, the broker could clean up the resources created on behalf of the client, etc.).

Heart-beating is negotiated between the client and the broker when the client connects to the broker (by sending a `CONNECT` frame) and the broker accepts the connection (by sending a `CONNECTED` frame to the client). Both frames accept a `heart-beat` header whose value contains two integers separated by a comma:

```
CONNECT
heart-beat:<cx>,<cy>

CONNECTED:
heart-beat:<sx>,<sy>
```

Let's take a look at what this code means:

- `<cx>` The smallest number of milliseconds between heartbeats that the client guarantees. If it is set to `0`, the client will not send any heartbeat at all.
- `<cy>` The desired number of milliseconds between heartbeats coming from the broker. If it is set to `0`, the client does not want to receive any heartbeat.
- `<sx>` The smallest number of milliseconds between heartbeats that the broker guarantees. If it is set to `0`, the broker will not send any heartbeat.
- `<sy>` The desired number of milliseconds between heartbeats coming from the client. If it is set to `0`, the broker does not want to receive any heartbeat.

When the client is successfully connected to the STOMP broker (it has received the `CONNECTED` frame), it must determine the frequency of the heartbeats to send to the broker and the frequency of heartbeats coming from the broker.

The values that are used to determine the frequency of heartbeats sent to the broker are `<cx>` and `<sy>`. If `<cx>` is `0` (the client will send no heartbeat) or if `<sy>` is `0` (the broker does not expect any client heartbeats), there will be no client heartbeating at all. This means that the broker will not be able to test the health of the client connection. Otherwise, both server and broker expect to exchange client heartbeats. The frequency is then determined by the maximum value between the value guaranteed by the client, `<cx>`, and the value desired by the broker, `<sy>`. In other words, the client must send heartbeats at a frequency of at least `MAX(cx,sy)` milliseconds.

For the heartbeats sent by the broker to the client, the algorithm is the same but uses the `<cy>` and `<sx>` values.

Let's take a simple example to illustrate the algorithm. We have a STOMP client that connects to the broker with the `heart-beat` header set to `0,60000` (the client will not send any heartbeats, but desires to receive the broker's heartbeats every minute):

```
CONNECT
heart-beat:0,60000
....
```

The broker accepts the connection and replies with a `CONNECTED` frame that contains a `heart-beat` header set to `20000,30000` (the broker guarantees to send heartbeats every 20 seconds and desires to receive the client's hearbeats every 30 seconds):

```
CONNECTED
heart-beat:20000,30000
....
```

Because the client specified that it will send no heartbeat (`0` as the first value of the `CONNECT`'s `heart-beat` header), client heartbeating is disabled and the broker should not expect any (even though it *desired* to receive them every 20 seconds).

The client desired to receive the broker heartbeat every minute (`60000` as the second value of the `CONNECT`'s `heart-beat` header). The broker replied that it can guarantee to send them at least every 30 seconds (second value of the `CONNECTED`'s `heart-beat` header). In that case, the broker and the client agrees that the broker must send heartbeats every minute (the maximum between 1 minute and 30 seconds). In other words, the broker *could* send heartbeats every 30 seconds (as it guaranteed in the `CONNECTED` frame), but the client will only check them every minute.

ActiveMQ supports heart-beating and mirrors the heartbeat values sent by the STOMP client. This lets the STOMP client be the sole decider of the heart-beating values.

This means that if a client connects with a `heart-beat` header set to `<cx>,<sy>`, the broker will accept the connection with a `heart-beat` header set to `<sy>,<cx>`.

The client guaranteed to send hearbeats every `<cx>` milliseconds, so the broker replied that it desires to receive them at this rate. The client desired to receive heartbeats every `<sy>` milliseconds, so the broker replied that it guarantees to send its heartbeat at this rate.

The client should set its `heart-beat` header according to its usage. For example, if an application is sending messages at a regular rate (such as the `Locations` iOS application), there is no need to send heartbeats to the broker at a similar (or faster) rate. The messages sent are proof enough of the client activity. Likewise, if a client expects to receive messages at a regular rate (such as the Locations web application), there is no need to require the broker to send frequent heartbeats.

However, if the application does not send messages often (the Locations web application will seldom sent text messages to the device' text topics), it probably should send heartbeats more frequently to inform the broker of its healthiness. Likewise, if the application does not receive messages often (such as the Locations iOS application), it should desire more frequent heartbeats from the broker.

StompKit Example

A `STOMPClient` supports heart-beating by passing the `heart-beat` header when it connects to the broker using its `connectWithHeaders:completionHandler` method.

By default, `StompKit` defines a heartbeat of `5000,10000` (send heartbeats every 5 seconds and receive them every 10 seconds).

Let's add heart-beating to the Locations iOS application. The application sends messages often (every time the device GPS position is updated), but receives them less frequently (when a user sends a message from the web application). We will guarantee to send heartbeats every minute (60000ms) and desire to receive them from the broker every 20 seconds (20000ms):

```
- (void)connect
{
    NSLog(@"Connecting...");
    self.client.errorHandler = ^(NSError* error) {
        NSLog(@"got error from STOMP: %@", error);
    };
    // will send a heartbeat at most every minute.
    // expect broker's heartbeat at least every 20 seconds.
    NSString *heartbeat = @"60000,20000";
    [self.client connectWithHeaders:@{ @"client-id": self.deviceID,
                                       kHeaderHeartBeat: heartbeat }
                  completionHandler:^(STOMPFrame *connectedFrame,
                                      NSError *error) {
                      ...
                  }];
}
```

stomp.js Example

The `STOMP` client has a `heartbeat` property composed of two properties:

- `heartbeat.outgoing` is the guaranteed frequency of heartbeat it can send to the broker (i.e., <cx>)
- `heartbeat.incoming` is the desired frequency of heartbeat coming from the broker (i.e., <cy>)

By default, stomp.js defines a heartbeat of `10000,10000` (to send and receive heartbeats every 10 seconds).

These properties must be modified *before* the `connect` method is called to take them into account:

```
// create the STOMP client
client = Stomp.client(url);

// will send a heartbeat at most every 20 seconds
client.heartbeat.outgoing = 20000;
// expects broker's heartbeat at least every minute
client.heartbeat.incoming = 60000;
client.connect({}, function(frame) {
  ...
});
```

Summary

In an ideal world, only the basic features of STOMP would be required to use messaging in mobile and web applications. However, to handle errors that will eventually happen under normal use, we need to leverage advanced STOMP features.

In this chapter, we learned to use:

- *Authentication* to ensure that only authenticated clients can communicate with the STOMP broker
- *Acknowledgment* to let the client explicitly accept the delivery of a message
- *Transaction* to send messages as a single atomic unit of work
- *Error handling* to face unexpected issues and eventually reconnect to the broker
- *Receipt* to receive confirmation that a frame has been succesfully processed by the broker
- *Heart-beating* to ensure that the network connection between the client and broker is healthy (and to ensure that it will kill the connection if that is not the case)

CHAPTER 5
Beyond STOMP

STOMP provides a simple yet flexible messaging protocol. It also offers an extensible way for brokers to provide additional features beyond the ones specified in the protocol.

In this chapter, we will show how to leverage broker features with STOMP headers. Until this chapter, all STOMP brokers could be used to send and receive messages from our applications; but in this chapter, you will have to check your broker documentation to see if it provides these features (or others not covered by this chapter).

Message Persistence

Some STOMP brokers support *persistent* messages to ensure that if a message is held by the broker when it crashes, the message will be persisted (which means stored on a durable support) so that the broker can fetch it when it restarts and handle it again. This prevents data loss (at the cost of performance, because the broker must ensure that the message is effectively written on the storage support). To use persistent messages, most STOMP brokers (including ActiveMQ) require that the message be sent with a `persistent` header set to `true`.

In the Locations application, we send two types of messages: one for the device location and one for text messages.

The location messages do not need to be persisted and survive a broker crash. There are minimal consequences if these messages are lost if the broker crashed. Once the broker is up again, the device will send an updated position.

However, for the text messages sent to the devices, we want to make sure that they are not lost before the device had a chance to receive it. We could declare these messages as persistent in the Locations web application to ensure that they survive a broker crash:

```
var order = "XXX";
client.send("/topic/mytopic", { persistent: true}, order);
```

When the broker receives the message, it will check the `persistent` header and persist the message if it is true.

Persisting messages adds a significant cost: the broker must write the message to a durable support (on a filesystem or in a database) and wait for the operating system to effectively write it (some operating systems will buffer data before writing them to disk). The broker must also have sufficient space on the durable support to write the persistent messages (once a persistent message has been delivered to all its consumers, the broker can safely discard it from the durable support).

To add further reliability, the client could use receipts (as described in "Receipts" on page 84) to wait until the broker has processed its message (and persisted it). When the receipt is received, the client is guaranteed that the message would survive a server crash. Without a receipt, there is a small window for failure if the server crashes *after* receiving the message but *before* storing it on its durable support.

Filtered Consumer

There are cases in which a consumer may be interested only by a subset of all messages delivered from a destination.

Every time a consumer is delivered a message, it can filter it out (using the message's headers or its payload) to take into account only the interesting ones. However, this is inefficient, because the consumer is still receiving all messages from the destination but may potentially discard many of them.

Some STOMP brokers allow you to specify a filter (or selector) when a consumer subscribes to a destination. Before delivering the message to the destination's consumers, the STOMP broker will check whether the message matches the consumer's filter. The message is delivered to the consumer only if it matches the filter.

A filter is similar to a SQL 92 conditional query using the message headers. The message payload is opaque for the STOMP broker and cannot be used by filters.

It turns out that this can be quite powerful, because STOMP allows us to add any user-specified header. If we want to filter out messages, we can put interesting information in the headers (either by removing them from the payload or duplicating them).

For example, we could add a `country` header when a location message is sent by the Locations iOS application. The value of this header would correspond to the country where the device is located (e.g., `FR` for France, `DE` for Germany, `IT` for Italy, etc.):

```
- (void)sendLocation:(CLLocation *)location
{
    // ...

    NSString *country = [self findCountryFrom:location];
    NSDictionary *headers = @{
        @"content-type": @"application/json;charset=utf-8",
        @"country": country
    };

    // send the message
    [self.client sendTo:destination
                headers:headers
                   body:body];
}
```

With this new header in the message representation, the consumers can now filter out messages to receive only those for certain countries.

If we want to receive only messages from France, Germany, and Italy, we just need to add a `selector` header when the consumer subscribes to the destination:

```
client.subscribe(destination, function(message) {
  // Only messages with the country header with the value FR, DE, or IT will
  // be delivered to the client.
  // ...
}, {selector: "country IN ('FR', 'DE', 'IT')"});
```

If the STOMP broker support filters consumers and your application can leverage them, this can significantly reduce the amount of messages sent on the network, saving bandwidth, CPU, and battery usage on the client (filtered out messages will not be sent over the network and will not be processed at all by the client).

ActiveMQ has specific rules about how to use a filtered consumer with STOMP, which are described in its Selectors documentation page.

Priority

Messages sent to destinations are usually delivered to a consumer in the order of their arrival. This means that if a producer sent messages A, B, and C in that order, a single consumer will receive the messages in the same order: A, B, and C.

However, there are cases where a producer wants to send a more *urgent* message that should take precedence over messages that it already sent.

Some STOMP brokers provide a way to achieve this using message *priority*. When a producer sends a message, it can set a `priority` header to change the message priority. The usual semantics for the priority value are copied from the JMS API, which defines ten levels of priority value, with `0` as the lowest priority and `9` as the highest (and `4` as the default priority).

A broker will often try to deliver expedited messages before messages of lower priority, but this is not a strong requirement. To ensure a *fair* delivery of messages, it may deliver lower priority messages so that a single producer sending only messages with the highest priority does not prevent the delivery of messages of lower priority from other producers.

In the Locations application, we do not have a use case where changing the priority of a message would make sense, because we use different channels for tracking device location and sending text messages to the device. If we were using a single channel for both kind of messages, we could give the text messages a higher priority (e.g., `7`) so that they could be delivered before messages with normal priority:

```
client.send(destination, { priority: 7 }, text);
```

Expiration

Messages can be valid for only a given duration, after which they are no longer relevant. For example, the locations messages sent by the Locations iOS application are valid for a short period of time, after which the device has likely moved to another location.

Some STOMP brokers allow messages to expire after a period of time. The broker will periodically check if messages held by its destinations have expired. If that is the case, it will discard them from the destinations and consumers will not receive them.

ActiveMQ accepts an `expires` header when a message is sent by a STOMP producer to specify the time until which the message is valid. The value is the number of milliseconds between the UNIX time (00:00:00, Thursday, January 1, 1970 UTC) and the expiration date.

For example, if we want to expire messages 10 minutes after they are sent by the `Locations` applications, we need to add an `expires` header with a value that is the number of milliseconds since the Unix time and ten minutes after now:

```
- (void)sendLocation:(CLLocation *)location
{
    // ...
```

```
        // 10 minutes from now
        NSTimeInterval expiration = [[NSDate date] timeIntervalSince1970] + 600000;
        NSDictionary *headers = @{
            @"content-type": @"application/json;charset=utf-8",
            @"expires": [NSNumber numberWithLong:(long)expiration]
        };

        // send the message
        [self.client sendTo:destination
                    headers:headers
                       body:body];
    }
```

Expiring messages can improve the health of your applications. Producers have no knowledge of when their messages will be consumed and by whom. However, if they know that the data sent in their messages has a limited lifetime, they can expire them after a given time instead of letting the broker deliver them to consumers after they stop being valid.

Summary

In this chapter, we learned that STOMP is a simple and flexible protocol that can be extended by brokers and clients using additional headers.

Based on the ActiveMQ broker, we see that STOMP can be extended to:

- Support *persistent* messages by passing a `persistent` header set to `true` when messages are sent.
- Create a *filtered consumer* use a `selector` filter to receive only messages with headers that match the filter.
- Send messages with a higher or lower priority using the `priority` header.
- Priority messages after an expiration date so that broker will not deliver messages after this date.

Depending on the STOMP broker you use, you may be able to use these features or others to improve the design of your architecture and reduce the network bandwidth and the CPU and battery usage so that producers and consumers only deal with relevant messages and ignore the others.

PART II
MQTT

This book covers the latest released version of the protocol at the time of this writing: MQTT 3.1.

MQTT is a simple and lightweight messaging protocol that only supports the publish/subscribe messaging model.

MQTT is a binary protocol and its wire format (the bits that are sent over the wire) is very compact, reducing bandwidth and memory usage. It defines a small set of features and there are clients for it on many platforms ranging from traditional operating systems to embedded devices. In Chapters 6 and 7, we will write a mobile application for iOS and a web application, but the description of the MQTT protocol applies to any client (although their API may differ).

For the MQTT Motions application, we will not run an MQTT broker on our local network, but instead use a public MQTT broker hosted by *iot.eclipse.org*, the portal for the Internet of Things development of the Eclipse foundation. This public broker is running using the Mosquitto project (the same codebase underneath MQTTKit) and any MQTT client can connect to it. While it is not appropriate to use it for production (there is no uptime guarantee for it), it shows that it is possible to use MQTT on the Internet.

MQTT clients can connect to it using the *iot.eclipse.org* hostname on the port 1883 (which is reserved for MQTT).

CHAPTER 6

Mobile Messaging with MQTT

In this chapter, we will write an iOS application using the MQTTKit library to send and receive messages using the MQTT protocol.

In "Motions Application Using MQTT" on page 9, we described the Motions application (Figure 6-1). In this chapter, we will write the iOS application that broadcasts the device motion data and receives alert messages.

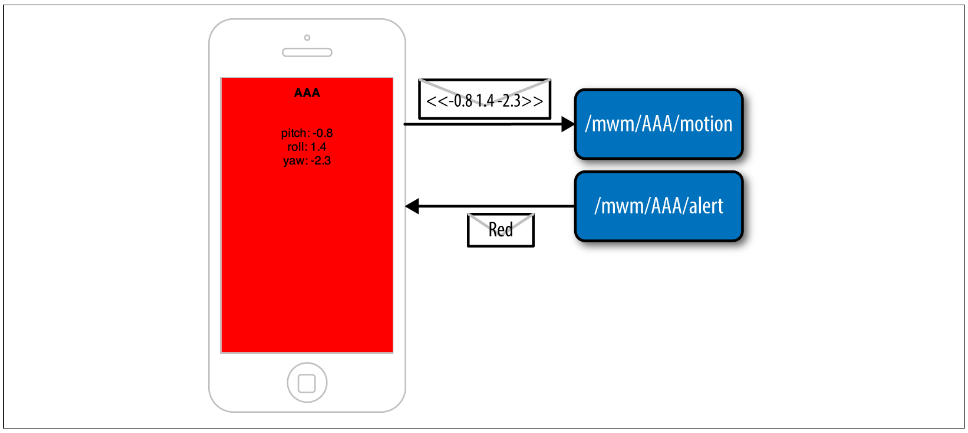


Figure 6-1. Diagram of the Motions iOS application

Throughout the chapter, we will show all the code required to run the application.

The whole application code can be retrieved from the GitHub repository in the *mqtt/ios/* directory.

MQTTKit

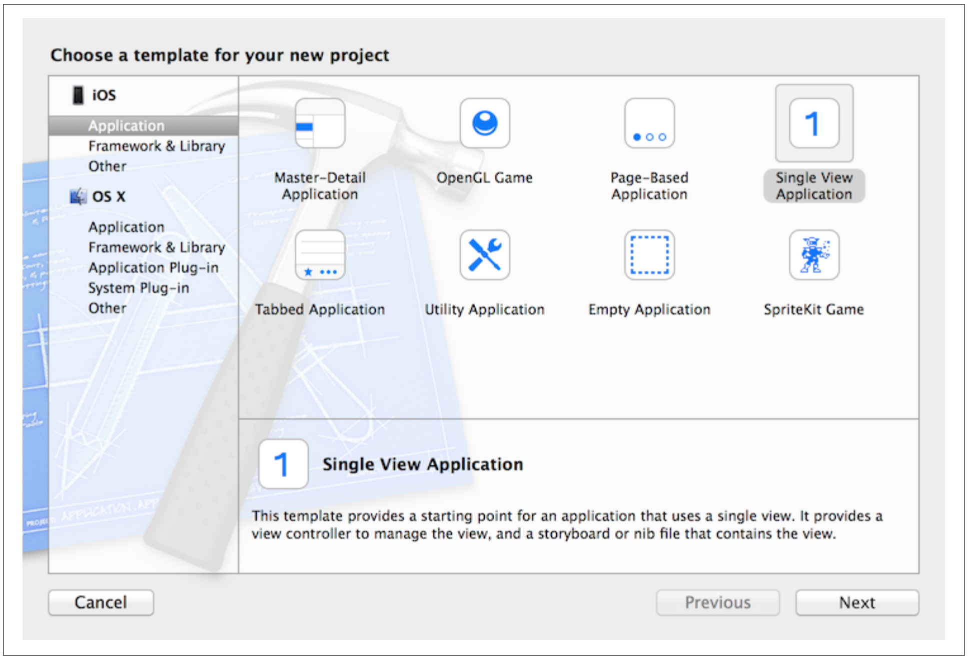
This book uses the MQTTKit library for iOS (and Mac OS X) applications. It is a modern Objective-C library that uses ARC and blocks to write messaging clients in a simple fashion. It is based on the Mosquitto project that provides a lower-level C implementation of MQTT.

The source code of this library project is hosted on GitHub.

Create the Motions Project with Xcode

We will use Xcode to create the Locations iOS application.

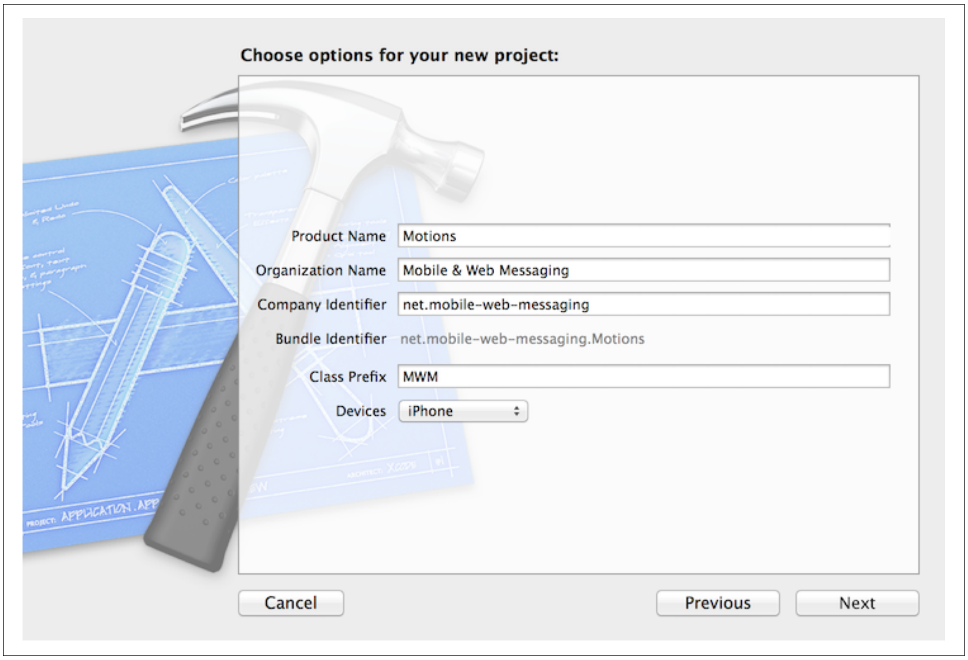
After Xcode is installed and started, we select "Create a new Xcode project" from its launch screen. The application will be composed of a single view, so we select the Single View Application template in iOS → Application from the template screen, as illustrated in Figure 6-2.



This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

Figure 6-2. Select Single View Application from the template screen

We will call the project Motions and select to build it only for iPhone devices, as illustrated in Figure 6-3.



Product Name: Motions
Organization Name: Mobile & Web Messaging
Company Identifier: net.mobile-web-messaging
Bundle Identifier: net.mobile-web-messaging.Motions
Class Prefix: MWM
Devices: iPhone

Figure 6-3. XCode project options screen

Finally, we will save it in a folder on our machine.

Create the Podfile

We will again use CocoaPods to manage the project dependencies (as explained in "Create the Podfile" on page 19 for the STOMP example).

We close Xcode because CocoaPods will modify the project settings to import the dependencies.

We create a file named *Podfile* at the root of the project (in the same directory as *Motions.xcodeproj*):

```
xcodeproj 'Motions.xcodeproj'

pod 'MQTTKit', :git => 'https://github.com/mobile-web-messaging/MQTTKit.git'

platform :ios, '5.0'
```

After saving this file, run the **`pod install`** command:

Example 6-1. Install Motions dependencies

```
$ pod install
Analyzing dependencies
Pre-downloading:    `MQTTKit`    from    `https://github.com/mobile-web-messaging/
MQTTKit.git`
Downloading dependencies
Installing MQTTKit (0.1.1)
Generating Pods project
Integrating client project

[!] From now on use `Motions.xcworkspace`.
```

We can now open Xcode again, but we must do it using the *workspace* file named *Motions.xcworkspace*, and not the *project* file named *Motions.xcodeproj* (Figure 6-4).

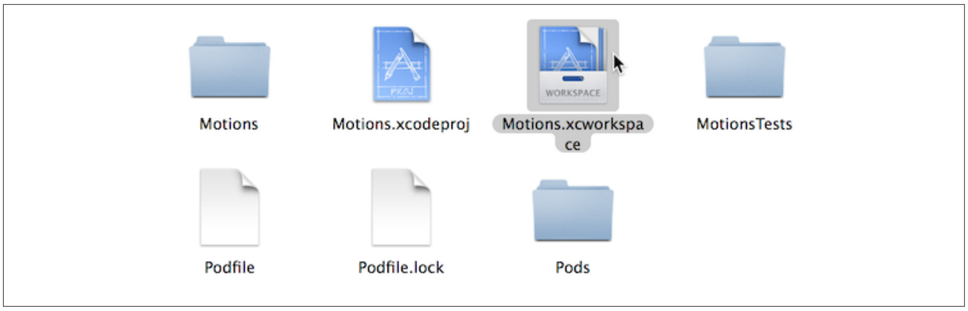


Figure 6-4. Open the workspace file

First, we will verify that the project is set up correctly and that the application can run in the iOS simulator.

We will simulate the latest iPhone devices by selecting Product → Destination → iPhone Retina (4-inch 64-bit) from the Xcode menu bar.

If we run the application by selecting Product → Run (or pressing ⌘+R), the iOS simulator starts and opens the application, which is composed of a blank view. This confirms that the project and its dependencies are successfully compiled and launched.

Identify the Device

This step is similar to the Locations iOS application described in "Identify the Device" on page 22.

We will generate a unique identifier for the iOS device and display it in the view.

Click on *Main.storyboard* to open it. From the Object library, drag a Label on the View's window. Place it at the top of the view and change the text to "Device ID," as shown in Figure 6-5.

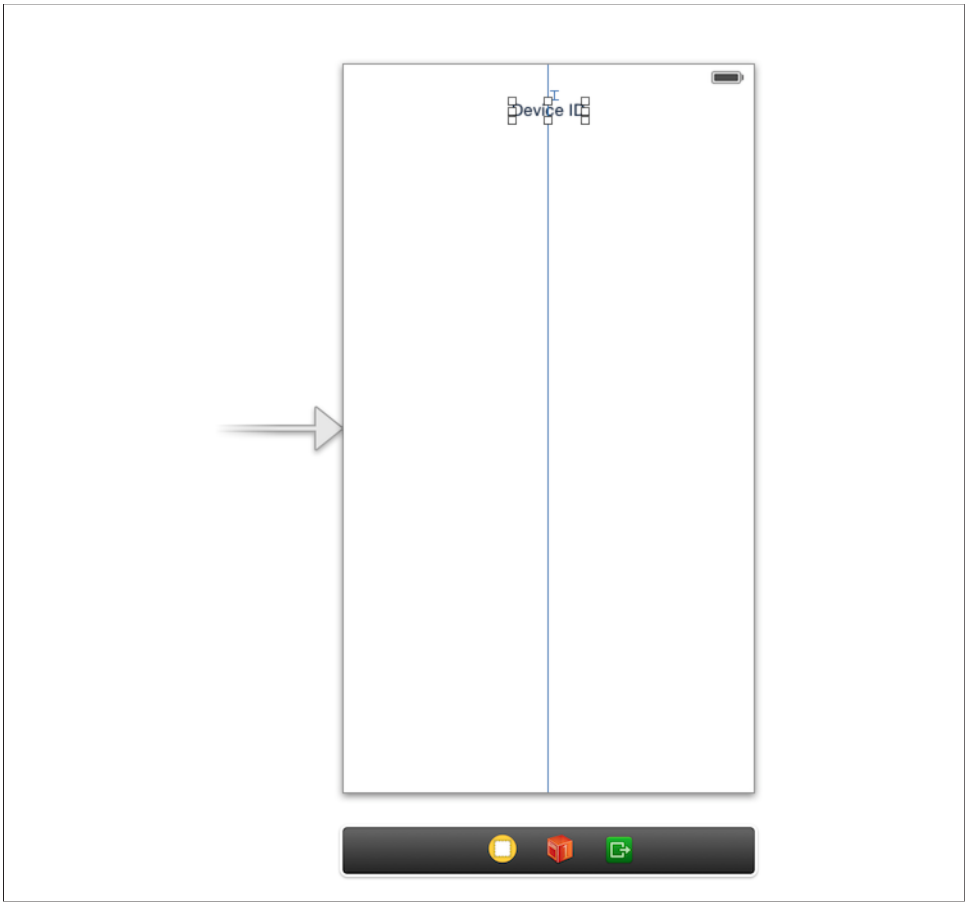


Figure 6-5. Add the Device ID label

We will again change its Font to System 13.0 and its Alignment to centered to fit the screen.

This label will be connected to a `deviceIDLabel` outlet property defined in the `MWMViewController` private interface in the *MWMViewController.m* file. We also add a `deviceID` string to store the device identifier:

```
@interface MWMViewController ()

@property (weak, nonatomic) IBOutlet UILabel *deviceIDLabel;

@property (strong, nonatomic) NSString *deviceID;

@end
```

Open the *Main.storyboard* and Ctrl-click on View Controller to see its connection panel. Drag from deviceIDLabel to the UILabel to connect it. See Figure 6-6.

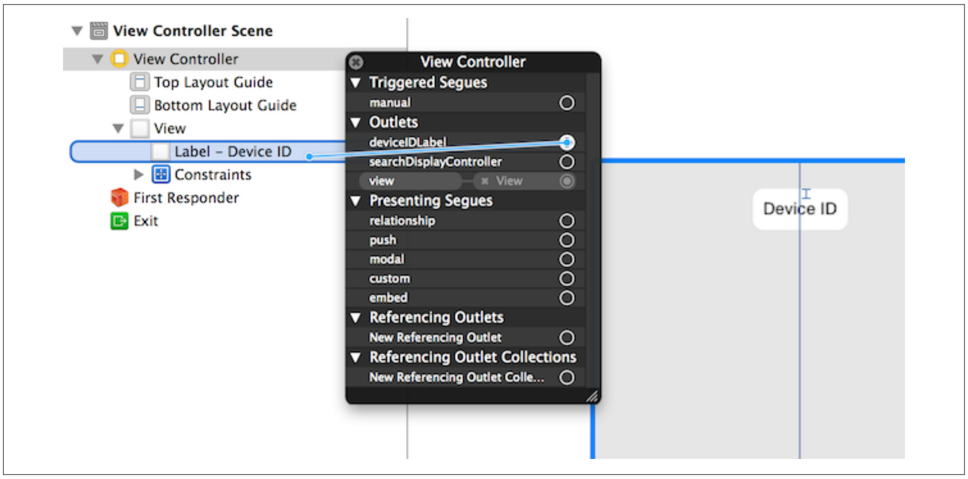


Figure 6-6. Connect the deviceIDLabel outlet property to the Device ID UILabel

The device identifier is generated in the `MWMViewController` implementation when the view is loaded and stored in the `deviceID` property. We also set the `deviceIDLa bel`'s `text` to this identifier:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.deviceID = [UIDevice currentDevice].identifierForVendor.UUIDString;
    NSLog(@"Device identifier is %@", self.deviceID);
    self.deviceIDLabel.text = self.deviceID;
}
```

Display the Device Motions Values

The device motion will be identified using the pitch, roll, and yaw values. To have some graphical feedback as we move the device, we will add three `UILabels` that show these three values.

Click on *Main.storyboard* to open it. From the Object library, drag three Labels on the View's window below the Device ID label. Change their respective text to "pitch," "roll," and "yaw," as illustrated in Figure 6-7.

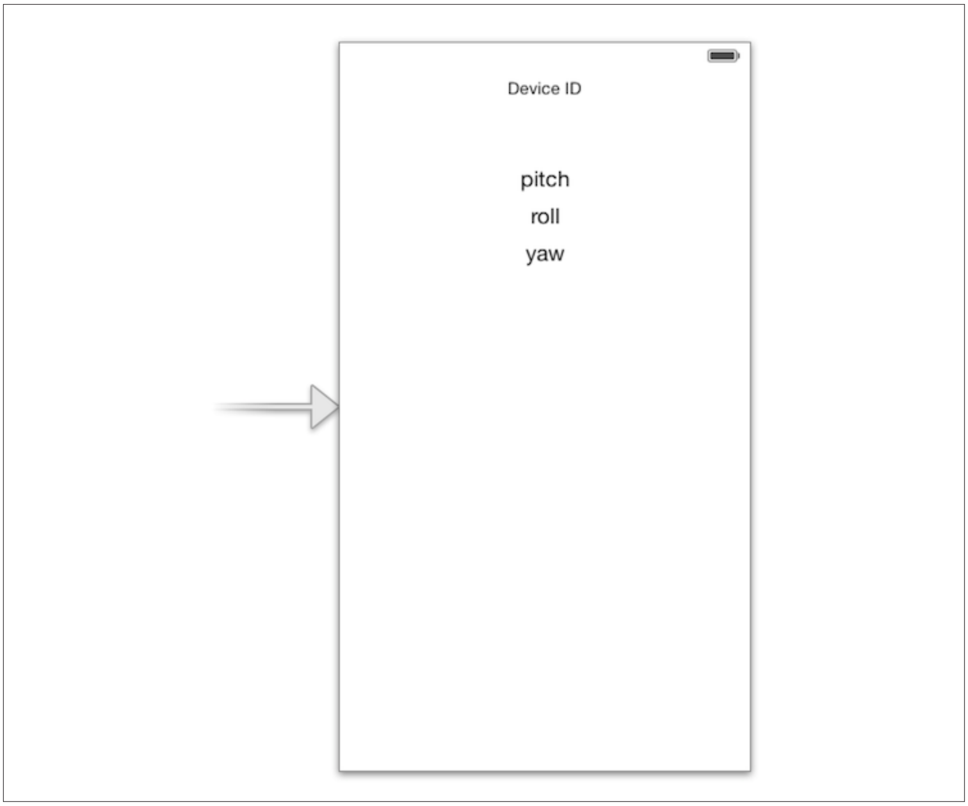


Figure 6-7. Add three labels to display the device's pitch, roll, and yaw values

We create three outlet properties in the `MWMViewController` private interface for these labels:

```
@interface MWMViewController ()

@property (weak, nonatomic) IBOutlet UILabel *deviceIDLabel;
@property (weak, nonatomic) IBOutlet UILabel *pitchLabel;
@property (weak, nonatomic) IBOutlet UILabel *rollLabel;
@property (weak, nonatomic) IBOutlet UILabel *yawLabel;

@property (strong, nonatomic) NSString *deviceID;

@end
```

The next step is to connect the three labels in the *Main.storyboard* to these three outlet properties.

Open the *Main.storyboard* and Ctrl-click on View Controller to see its connection panel. Drag from its pitchLabel property to the corresponding pitch UILabel to connect it (see Figure 6-8).

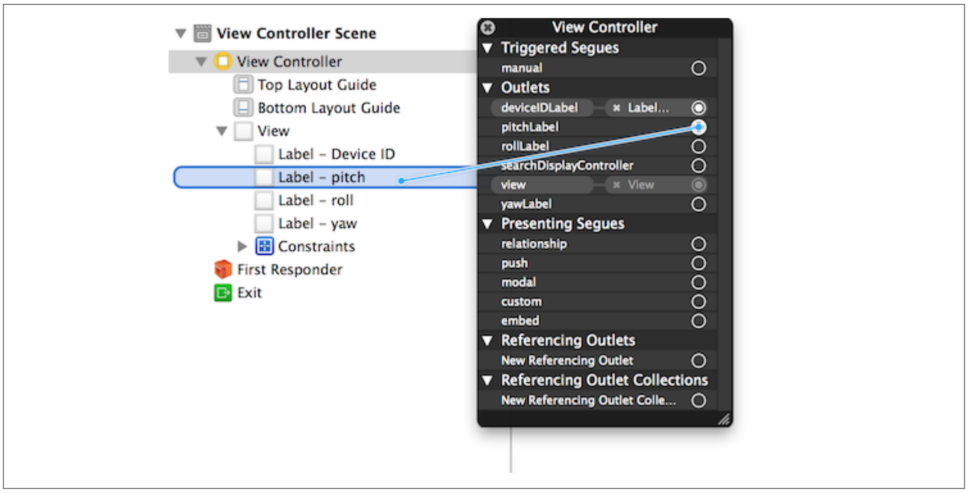


Figure 6-8. Connect the pitchLabel outlet property to the pitch UILabel

Repeat this operation for the `rollLabel` and `yawLabel` to connect them.

At this stage, the graphical objects are connected and we can capture the device motion to update these labels and then broadcast the motion data using MQTT.

Capture the Device Motions with CoreMotion Framework

iOS provides the `CoreMotion` framework to capture the motion of the devices.

We need to add it to the libraries linked by the application (see Figure 6-9). Click on the Motions project and then the Motions target. In the General tab, under the Linked Frameworks and Libraries section, click on the + button. In the selection window, type "CoreMotion," select CoreMotion.framework, and click on the Add button.

We can now use the CoreMotion framework by importing *CoreMotion/CoreMotion.h* at the top of the *CoreMotion/CoreMotion.h* file.

We will also define a `motionManager` property in the `MWMViewController` private interface to use `CoreMotion`:

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

@interface MWMViewController ()

@property (strong, nonatomic) CMMotionManager *motionManager;

@end
```

This `motionManager` is used to capture the device motions. We must create a new `CMMotionManager`, specify the interval of updates, and call its `startDeviceMotionUp` `datesToQueue:withHandler:` method to get the device motion periodically in a block. We create a new `NSOperationQueue` to receive these updates on this queue.

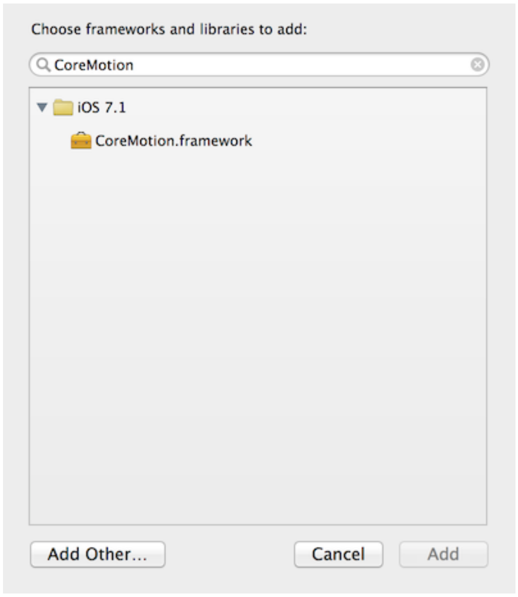


Figure 6-9. Add the CoreMotion framework

The device motion is represented by a `CMDeviceMotion` object. In our example, we are interested only in its `attitude` property that contains the pitch, roll, and yaw value we want to broadcast. Their values are expressed in radians, so we will convert them in degrees to display them.

The block to receive motion updates is executed on the `NSOperationQueue` we created, so we cannot update the `UILabel` from it. We must instead create another block and call `dispatch_async` to execute the graphical changes on the UI main queue (which is retrieved by calling `dispatch_get_main_queue()`).

All this logic can be written in `viewDidLoad` so that the motion manager starts receiving updates when the view is loaded:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.deviceID = [UIDevice currentDevice].identifierForVendor.UUIDString;
    NSLog(@"Device identifier is %@", self.deviceID);
    self.deviceIDLabel.text = self.deviceID;

    self.motionManager = [[CMMotionManager alloc] init];
    // get the device motion updates every second.
    self.motionManager.deviceMotionUpdateInterval = 1;
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [self.motionManager startDeviceMotionUpdatesToQueue:queue
                                        withHandler:^(CMDeviceMotion *motion,
                                                      NSError *error) {
        if(!error) {
            CMAttitude *attitude = motion.attitude;
            dispatch_async(dispatch_get_main_queue(), ^{
                // convert values from radians to degrees
                double pitch = attitude.pitch * 180 / M_PI;
                double roll = attitude.roll * 180 / M_PI;
                double yaw = attitude.yaw * 180 / M_PI;
                self.pitchLabel.text =
                    [NSString stringWithFormat:@"pitch: %.0f°", pitch];
                self.rollLabel.text =
                    [NSString stringWithFormat:@"roll: %.0f°", roll];
                self.yawLabel.text =
                    [NSString stringWithFormat:@"yaw: %.0f°", yaw];
            });
        }
    }];
}
```

We also need to notify the `motionManager` that we no longer want to receive updates when the view is no longer used. We need to call its `stopDeviceMotionUpdates` method inside the view controller's `dealloc` method:

```
- (void)dealloc
{
    [self.motionManager stopDeviceMotionUpdates];
}
```

At this stage, if you run the Motions application on your iPhone and move it, the pitch, roll, and yaw labels will be updated to reflect the changes in the device motions (Figure 6-10).

The iOS Simulator cannot simulate device motions. If you run the Motions application in the simulator, the `motionManager` will not send any device motions updates. At the time of writing this book, the only way to test this code is to run the application on a real iOS device.

We now capture the device motions and display them. The next step is to broadcast them by sending MQTT messages.

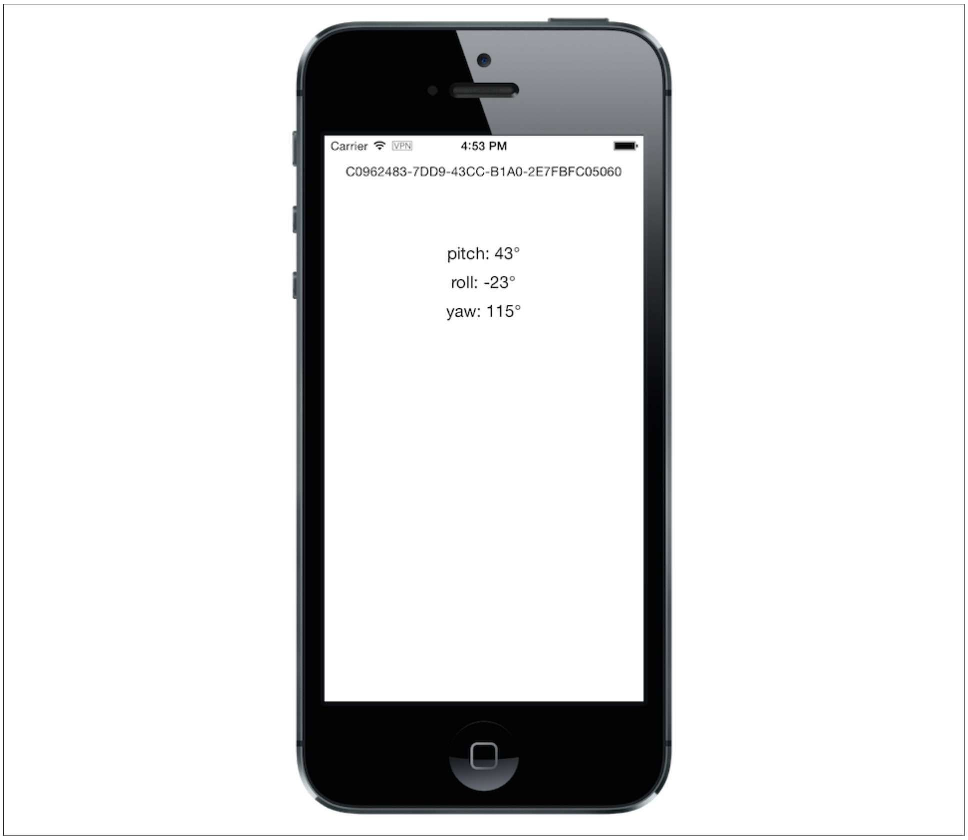


Figure 6-10. The motion values change when the device moves

Create an MQTT Client with MQTTKit

To send and receive messages with MQTT, we must first import the MQTTKit library that was added to the project using CocoaPods at the beginning of this chapter.

We must import its header file *MQTTKit.h* at the top of the *MWMViewController.m* file and add an `MQTTClient` property named `client` to the `MWMViewController` private interface.

We also define a constant to represent the hostname of the MQTT broker we are using in *iot.eclipse.org*:

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

#define kMqttHost @"iot.eclipse.org"

@interface MWMViewController ()
```

```
@property (strong, nonatomic) MQTTClient *client;

@end
```

We will create a new `MQTTClient` object in the `MWMViewController`'s `viewDidLoad` method. An `MQTTClient` must be uniquely identified for the MQTT brokers it connects to. We can use the `deviceID` as its client identifier:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    ...

    self.client = [[MQTTClient alloc] initWithClientId:self.deviceID];
}
```

Connect to an MQTT Broker

An MQTTKit client will connect to the MQTT Broker when its `connectToHost:completionHandler:` method is called. MQTTKit is event-driven, so the client will be *effectively* connected when its `completionHandler` block is called and the return code `MQTTConnectionReturnCode` is equal to `ConnectionAccepted`.

You cannot assume that the client is connected when the `connectToHost:completionHandler:` method returns. Any actions that require the client to be connected must happen inside the `completionHandler` block.

We will encapsulate this code in a `connect` method:

```
#pragma mark - MQTTKit Actions

- (void)connect
{
    NSLog(@"Connecting to %@...", kMqttHost);
    [self.client connectToHost:kMqttHost
             completionHandler:^(MQTTConnectionReturnCode code) {
        if (code == ConnectionAccepted) {
            NSLog(@"connected to the MQTT broker");
        } else {
            NSLog(@"Failed to connect to the MQTT broker: code=%lu", code);
        }
    }];
}
```

We will call this method from `viewDidLoad` to connect to the MQTT broker as soon as the view is loaded:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
```

```
        self.deviceID = [UIDevice currentDevice].identifierForVendor.UUIDString;
        NSLog(@"Device identifier is %@", self.deviceID);
        self.deviceIDLabel.text = self.deviceID;

        ...

        self.client = [[MQTTClient alloc] initWithClientId:self.deviceID];
        [self connect];
    }
```

Disconnect from an MQTT Broker

The `client` can disconnect from the MQTT broker by calling its `disconnectWithCompletionHandler:` method.

The `completionHandler` block as a `code` parameter will be `0` if the disconnection succeeds:

```
    - (void)disconnect
    {
        [self.client disconnectWithCompletionHandler:^(NSUInteger code) {
            if (code == 0) {
                NSLog(@"disconnected from the MQTT broker");
            } else {
                NSLog(@"disconnected unexpectedly...");
            }
        }];
    }
```

We want to disconnect from the MQTT broker when the `MWMViewController` is no longer used. We will call the `disconnect` method from `dealloc`:

```
    - (void)dealloc
    {
        [self.motionManager stopDeviceMotionUpdates];
        [self disconnect];
    }
```

Send MQTT Messages

The `MWMViewController` automatically connects to the MQTT broker when its view is loaded and disconnects when it is deallocated. The next step is to send messages every time the device motion values are updated.

The MQTT protocol is a binary protocol. The message payload must be encoded as binary data to be sent. The MQTTKit library provides two methods to send messages:

- The `publishData:toTopic:withQos:retain:` method expects an `NSData` object as the message payload and its `bytes` will be used.
- The `publishString:toTopic:withQos:retain:` method can also be used for the common case of sending a text message. Internally, the `NSString` that is passed in as a parameter is encoded as an `NSData` using the UTF-8 encoding.

In the Motions iOS application, we send a message with a binary payload composed of three 64-bit floats for the pitch, roll, and yaw values contained in a `CMAttitude` object. We will build the payload's `NSData` by converting the double values to a platform-independent format using the `CFConvertDoubleHostToSwapped` function.

The other three parameters to the `publish…` methods are the same for both the binary and text payload version.

The `topic` parameter is the name of the topic to send the message. According to "Messaging models for the Motions application" on page 11, the name of the topic is `/mwm/XXX/motion` where "XXX" is the device identifier.

The `qos` parameter corresponds to the *Quality of Service* (or QoS) to use to deliver the messages to the consumers.

Quality of Service

The MQTT protocol defines three levels of Quality of Service:

- `At Most Once` (with the value `0` represented by `AtMostOnce` in MQTTKit)
- `At Least Once` (with the value `1` represented by `AtLeastOnce` in MQTTKit)
- `Exactly Once` (with the value `2` represented by `ExactlyOnce` in MQTTKit)

These levels of QoS determine the guarantee that the MQTT broker will accept to deliver a message. With `At Most Once`, the MQTT broker guarantees that the published message will be delivered at most once to its consumers. This means that the consumers may not receive the message at all. If an error (such as a network failure or a crash) occurs while the message is sent to the broker, it is possible that it will be lost and the consumers will never receive it.

With `At Least Once`, the MQTT broker guarantees that the published message will be delivered at least once to the consumers. This also means that a consumer may receive the same message twice. If there is an error when the producer sends the message to the broker and a message acknowledgment has not been received, it will resend it a second time as a *duplicate* (the MQTT message will have a `DUP` bit set). When the broker receives this duplicate message, it will redeliver it to the consumer, but it is possible that they in fact received the original message. The consumer might need to check if the DUP bit is set on the delivered message to know whether it is an original message (and it must process it) or a duplicate (and it can discard it).

The At Least Once QoS offers the guarantee that no published message will be lost, but at the cost of performance and additional code on the consumer side. The performance cost is caused by the additional message (a PUBACK message) sent from the broker to the client to acknowledge that it has received the published message. That means that using this QoS level to publish N messages will involve exchanging 2*N messages between the producer and the broker.

The highest level of delivery is provided using the Exactly Once QoS. With that level, the MQTT broker guarantees that the published message will be delivered *exactly* once by the consumers. There will be no lost messages or duplicate messages. This is guaranteed by additional exchanges of messages between the producer and broker (PUBREC, PUBREL, PUBCOMP messages). That means that using this QoS level to publish N messages will involve exchanging 4*N messages, requires four times more network trips than the lowest level of QoS of At Most Once and twice more than the At Least Once level.

Choosing the correct QoS depends on the type of message exchanged and the *importance* of its payload. In the Motions iOS application, the published message contains device motion that is updated every second. It is acceptable if a published message is *lost* because a new message with updated content will be sent just one second later. Using the At Most Once QoS is the best choice for this type of message.

All the complexities of using a higher level of QoS are transparent from the application using MQTT, as it is the responsibility of the client library to handle it. However, you need to be aware of the cost associated with using these QoSes, as they can significantly impact your application performance and the device in general (the additional network trips will drain the battery life).

Retained Message

The final parameter of the publish… methods is a boolean to specify whether the published message must be *retained* by the topic.

If this flag is set on the message, the broker will deliver the message to its subscribers and keep holding the message. If a new consumer subscribes to this topic, the broker will deliver the retained message to it. This is useful, as the new subscriber will not have to wait for a publisher to send a message to receive new data. The retained message contains the *Last Known Good* value.

In our case, we will publish messages with retain set to YES. If consumers subscribe to the device motion topic *after* the device stops updating its motion values, they will still be able to use the last known device motion value. This example is a bit of a stretch. A more interesting example would be an application broadcasting its location (similar to the `Locations` application). Using retained message would allow the consumers to know the last known position of the device before it stops broadcasting its position.

To sum up, the `Motions` application will send a message:

- With a binary payload composed of three 64-bit floats for the device's pitch, roll, and yaw values
- To the device motion topic `/mwm/XXX/motion` where `XXX` is the device identifier
- With a QoS of `At Most Once`, because we accept that a published message may not be delivered
- With `retain` set to `YES` so that the broker will retain the Last Known Good message to deliver it to new subscribers

We will encapsulate this code in a `send:` method taking a `CMAttitude` parameter:

```
- (void)send:(CMAttitude *)attitude
{
    uint64_t values[3] = {
        CFConvertDoubleHostToSwapped(attitude.pitch).v,
        CFConvertDoubleHostToSwapped(attitude.roll).v,
        CFConvertDoubleHostToSwapped(attitude.yaw).v
    };
    NSData *data = [NSData dataWithBytes:&values length:sizeof(values)];
    NSString *topic =[NSString stringWithFormat:@"/mwm/%@/motion",
    self.deviceID];
    [self.client publishData:data
                     toTopic:topic
                     withQos:AtMostOnce
                      retain:YES
           completionHandler:nil];
}
```

The message will contain the motion values in radians. It will be up to the consumers to convert them in degrees if necessary.

Finally, the last step is to call this method every time a device motion value is updated by the `motionManger`. This occurs in the `viewDidLoad` method inside the `handler` block passed to the `motionManger`'s `startDeviceMotionUpdatesToQueue:withHandler:` method:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.deviceID = [UIDevice currentDevice].identifierForVendor.UUIDString;
    NSLog(@"Device identifier is %@", self.deviceID);
```

```
        self.deviceIDLabel.text = self.deviceID;

        self.motionManager = [[CMMotionManager alloc] init];
        // get the device motion updates every second.
        self.motionManager.deviceMotionUpdateInterval = 1;
        NSOperationQueue *queue = [[NSOperationQueue alloc] init];
        [self.motionManager startDeviceMotionUpdatesToQueue:queue
                                    withHandler:^(CMDeviceMotion *motion,
                                                  NSError *error) {
            if(!error) {
                CMAttitude *attitude = motion.attitude;
                dispatch_async(dispatch_get_main_queue(), ^{
                    // convert values from radians to degrees
                    double pitch = attitude.pitch * 180 / M_PI;
                    double roll = attitude.roll * 180 / M_PI;
                    double yaw = attitude.yaw * 180 / M_PI;
                    self.pitchLabel.text =
                        [NSString stringWithFormat:@"pitch: %.0f°", pitch];
                    self.rollLabel.text =
                        [NSString stringWithFormat:@"roll: %.0f°", roll];
                    self.yawLabel.text =
                        [NSString stringWithFormat:@"yaw: %.0f°", yaw];
                });
                [self send:attitude];
            }
        }];

        self.client = [[MQTTClient alloc] initWithClientId:self.deviceID];
        [self connect];
    }
```

We now have the `Motions` iOS application that is sending MQTT messages. How can we check that this is working as expected?

Conversely to `STOMP`, `MQTT` is a binary protocol and we cannot use a simple telnet client to create a consumer and receive messages sent by the application.

However, the Mosquitto broker provides a simple command-line tool to send and receive message from an MQTT broker. Appendix B explains how to download and install the Mosquitto broker. After it is done, we can use its `mosquitto_sub` command-line tool to connect to an MQTT broker (hosted at *iot.eclipse.org*) and subscribe to the device motion topic (in my case, `/mwm/C0962483-7DD9-43CC-B1A0-2E7FBFC05060/motion`; you will have to replace it using your own device identifier).

This tool will display the message payload. We are sending binary payload, so we will pipe the command into the `hexdump` tool to display the hexadecimal representation of the binary payload:

```
$ mosquitto_sub -h iot.eclipse.org -t /mwm/C0962483-7DD9-43CC-B1A0-2E7FBFC05060/
motion | hexdump
```

```
...
0000050 aa b0 4c 3f 9b 41 0c 6b 08 35 d3 3f d2 4b 23 f2
0000060 71 1e 47 0a 3f d5 05 6a c4 37 52 16 3f d8 f7 b5
0000070 34 f6 19 ea bf d2 97 6f 1a 65 86 af 0a 3f af 23
0000080 78 91 85 1c 8d bf df b9 12 c4 78 64 1c 3f cb 3c
0000090 50 fd 05 26 5b 0a 3f d1 60 87 16 0b 12 9e bf c2
...
```

This confirms that the Motions application is effectively publishing MQTT messages.

Receive MQTT Messages

As described in "Motions Application Using MQTT" on page 9, the Motions iOS application is also a consumer from the topic `/mwm/XXX/alert`. When it receives a message from this topic, it must change its background color to "alert" the user.

Let's write the method that alerts the user by changing the background color first. This `warnUser:` method takes an `NSString` parameter that should correspond to a color. Using `UIKit` animations, we will:

1. Animate the controller's view to change its background color from its original color to the one created from the `NSString` parameter.
2. Wait two seconds after the first animation is completed to revert back to the original background color:

```
# pragma mark - UI Actions
// Warn the user by changing the view's background color to the specified color
// during 2 seconds
- (void)warnUser:(NSString *)colorStr
{
    // keep a reference to the original color
    UIColor *originalColor = self.view.backgroundColor;

    [UIView animateWithDuration:0.5
                          delay:0.0
                        options:0
                     animations:^{
                         // change it to the color passed in parameter
                         NSString *colorCode =
                            [NSString stringWithFormat:@"%@Color", colorStr];
                         SEL sel = NSSelectorFromString(colorCode);
                         UIColor* color = nil;
                         if ([UIColor respondsToSelector:sel]) {
                             color  = [UIColor performSelector:sel];
                         } else {
                             color = [UIColor redColor];
                         }
                         self.view.backgroundColor = color;
                     }
                     completion:^(BOOL finished) {
                         // after a delay of 2 seconds, revert it to
                         // the original color
                         [UIView animateWithDuration:0.5
                                               delay:2
                                             options:0
                                          animations:^{
                                              self.view.backgroundColor =
                                                  originalColor;
                                          }
                                          completion:nil];
                     }];
}
```

To consume messages from an MQTT broker, the `client` must do the following:

1. Subscribe to its topic of interest.
2. Set its `messageHandler` property, which will be called every time a message is delivered.

Note that you can subscribe to many topics from the client but it has only one `messageHandler` property. If the client is subscribed to different topics, its `messageHandler` must determine the topic from which the message is consumed.

Subscription

The Motions application will subscribe to its device alert topic `/mwm/XXX/alert` by calling the method `subscribe:withQos:completionHandler:` on its `client` property.

The first parameter is the device alert topic. We will define it at the top of the *MWMViewController.m* file:

```
#define kAlertTopic @"/mwm/%@/alert"
```

The `subscribe:withQos:completionHandler:` method takes a `qos` parameter that corresponds to the level of quality of service at which the consumer wants to recieve messages from the topic.

The completion handler will be called when the client is effectively subscribed to the topic. The handler has a `grantedQos` parameter that corresponds to the effective quality of service. The producer is responsible for determining the maximum quality of service at which a message can be delivered, but the consumer can decide to *downgrade* the quality of service according to its usage. For example, a producer may publish a message with a QoS of `ExactlyOnce`, but a consumer may decide that it is acceptable if there are message duplicates and downgrade its QoS to `AtLeastOnce`.

In our case, we will request to have messages delivered with a `qos` set to `AtLeastOnce`, as we do not want to lose messages but can accept duplicate messages:

```
- (void)subscribe
{
    NSString *topic = [NSString stringWithFormat:kAlertTopic, self.deviceID];
    [self.client subscribe:topic
                   withQos:AtLeastOnce
         completionHandler:^(NSArray *grantedQos) {
        NSLog(@"subscribed to %@ with QoS %@", topic, grantedQos);
    }];
}
```

We will subscribe to the alert topic as soon as the client is connected to the MQTT broker by calling this subscribe method from inside the completionHandler in the connect method:

```
- (void)connect
{
    NSLog(@"Connecting to %@...", kMqttHost);
    [self.client connectToHost:kMqttHost
             completionHandler:^(MQTTConnectionReturnCode code) {
        if (code == ConnectionAccepted) {
            NSLog(@"connected to the MQTT broker");
            [self subscribe];
        } else {
            NSLog(@"Failed to connect to the MQTT broker: code=%lu", code);
        }
    }];
}
```

Unsubscribing

To unsubscribe from a topic and stop receiving messages from it, we will call the unsubscribe:withCompletionHandler: method of the client where the first parameter is the topic to unsubscribe from (the alert topic in our case). The second parameter is a completion handler that is called back when the client has been acknowledged by the server that it is effectively unsubscribed. We do not have any need for this information so we just pass nil as the handler:

```
- (void)unsubscribe
{
    NSString *topic = [NSString stringWithFormat:kAlertTopic, self.deviceID];
    [self.client unsubscribe:topic withCompletionHandler:nil];
}
```

We will call this unsubscribe method just before disconnecting from the MQTT broker from the dealloc method:

```
- (void)dealloc
{
    [self.motionManager stopDeviceMotionUpdates];
    [self unsubscribe];
```

```
        [self disconnect];
    }
```

Because we disconnect just after unsubscribing, we could skip that step and just disconnect from the MQTT broker. At that moment, the MQTT broker will automatically unsubscribe the client from any topic. However, it is a good practice to explicitly unsubscribe from the subscribed topic. There are also many cases where unsubscribing may occur at a different time than the disconnection. In these cases, we cannot rely on the client disconnection to perfom the unsubscription.

Define an MQTTMessage Handler

Subscribing to a topic is the first step in receiving messages with MQTTKit. The second step is to define a block that will be called every time a message is received from a subscribed topic.

The `client`'s `messageHandler` property defines an `MQTTMessageHandler` block. This block has an `MQTTMessage` parameter representing the MQTT message that is delivered to the client.

The `MQTTMessage` interface defines four properties corresponding to the message data:

- `mid` is a `unsigned short` corresponding to the *message ID*.
- `topic` is the name of the topic that this message is coming from. If the client is subscribed to many topics, we must use this property to determine the topic from which the received message is coming.
- `retained` is a `BOOL` to check whether the message was retained (and contains the last known good value) or not (in which case it is a *fresh* message).
- `payload` is an `NSData` object containing the binary content of the message payload.

Because sending and receiving text messages is very common, the `MQTTMessage` interface also defines a `payloadString` method that returns an `NSString` decoded from the message binary payload using UTF-8.

In the `Motions` application, we expect to receive a text payload and will use this `payloadString` to extract the color string from the received message.

We need to set the `client`'s `messageHandler` *before* subscribing to the alert topic so that we do not miss any alert message sent after we subscribe but *before* the `messageHandler` is defined. We will do that in the `viewDidLoad` method just after creating the `client` instance:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.deviceID = [UIDevice currentDevice].identifierForVendor.UUIDString;
    NSLog(@"Device identifier is %@", self.deviceID);
    self.deviceIDLabel.text = self.deviceID;

    ...

    self.client = [[MQTTClient alloc] initWithClientId:self.deviceID];
    // use a weak reference to avoid a retain/release cycle in the block
    __weak MWMViewController *weakSelf = self;
    self.client.messageHandler = ^(MQTTMessage *message) {
        NSString *alertTopic =
            [NSString stringWithFormat:kAlertTopic, weakSelf.deviceID];
        if ([alertTopic isEqualToString:message.topic]) {
            NSString *color = message.payloadString;
            dispatch_async(dispatch_get_main_queue(), ^{
                [weakSelf warnUser:color];
            });
        }
    };

    [self connect];
}
```

We extracted the color using the message `payloadString` method after checking that it was indeed coming from the device alert topic. We then call the `warnUser:` method in a block that is run on the main queue, because it contains code related to `UIKit`.

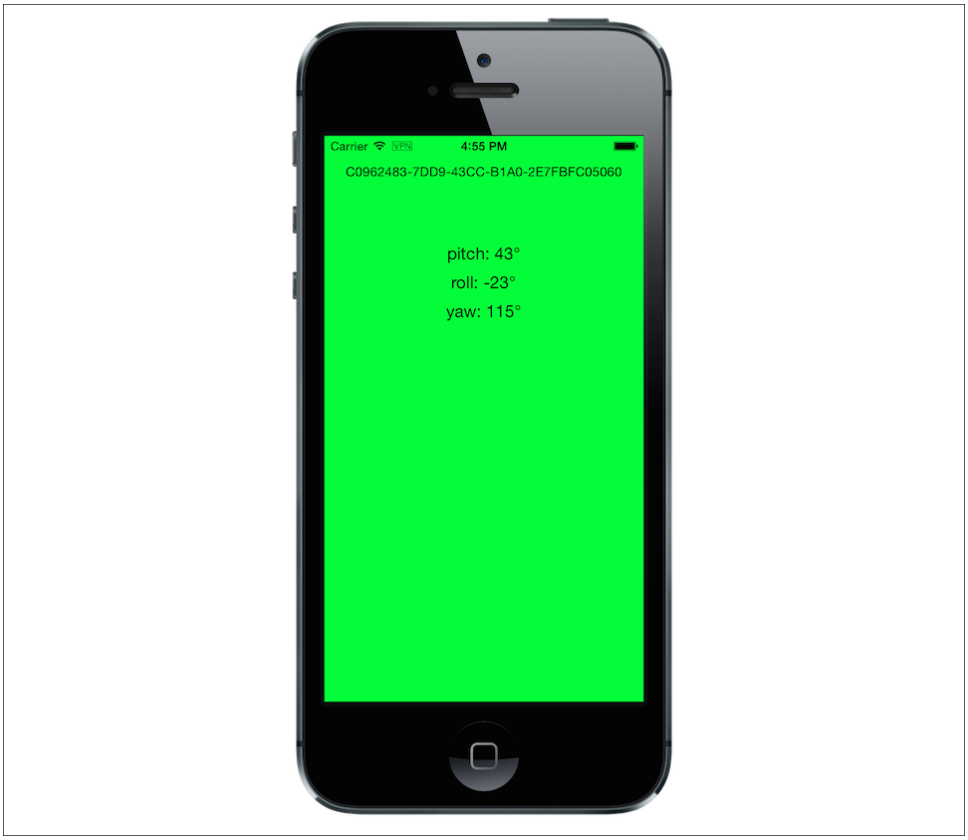
To avoid a retain/release cycle between `self` and the `messageHandler` block, we need to create a *weak* reference of `self` in order to use it inside the block.

How can we verify that the Motions application is effectively receiving alert messages? To verify that the application was sending messages, we used the `mosquitto_sub` tool. We will now use the opposite tool, `mosquitto_pub`, to publish a message on the alert topic and verify that the application background color changes.

The `mosquitto_pub` can send a text payload using the `-m` option. We will use this option to pass the background color (in this case, `green`):

```
$ mosquitto_pub -h iot.eclipse.org -t /mwm/C0962483-7DD9-43CC-B1A0-2E7FBFC05060/
alert -m green
```

After this message is sent, the device will receive it and change its background color to green (Figure 6-11).


Carrier VPN 4:55 PM

C0962483-7DD9-43CC-B1A0-2E7FBFC05060

pitch: 43°

roll: -23°

yaw: 115°

Figure 6-11. Alert message is received by the Motions iOS application

Summary

In this chapter, we learned to use MQTTKit to send and receive MQTT messages from an iOS application.

To send a message, the application must do the following:

1. Connect to the MQTT broker.
2. Send the message to the topic.

To consume a message, the application must do the following:

1. Connect to the MQTT broker.
2. Subscribe to the topic.

3. Define a message handler block that is called every time a message is received. This block is executed on a dispatch queue. If there is any code that changes the user interface, it must be wrapped in a block executed on the main queue.

We used two different types of message payloads:

- A binary payload to send the device motions values as three 64-bit floats.
- A text payload to extract a background color from the messages received on the alert topic.

CHAPTER 7

Web Messaging with MQTT

In this chapter, we will write a web application that sends and receives messages using the MQTT protocol over HTML5 Web Sockets (Figure 7-1).

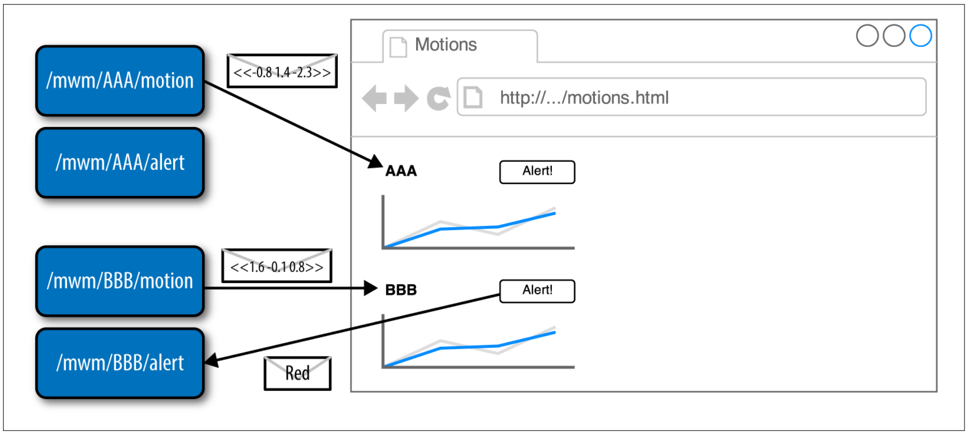


Figure 7-1. Diagram of the Motions web application that displays data for two devices

Throughout the chapter, we will show all the code required to run the application. The whole application code can be retrieved from the GitHub repository in the *mqtt/web/* directory.

Eclipse Paho JavaScript Client

Earlier, we mentioned that iot.eclipse.org provides a public MQTT broker that we use for our MQTT applications. Eclipse also provides a variety of MQTT clients for different languages and platforms. In particular, it has a JavaScript client for web browsers, which uses HTML5 Web Sockets.

The source code of the project is hosted in this Git repository.

Bootstrap the Motions Web Application

As we explained in "Motions Application Using MQTT" on page 9, this web application will display on a web page the motion data sent by the devices using the Motions iOS application written in . Additionally, the web application will also be able to send alert messages to the devices.

It will be a very simple one-page web application that can be run from a web server serving static pages. It does not require any server-side runtime as all the code will be executed inside the web browser using JavaScript.

The device motions data will be displayed graphically as Sparklines. The web application uses jQuery and a library named jQuery Sparklines to draw them.

Note, however, that we use jQuery for convenience, but the MQTT JavaScript client does not require it at all and can be used with any JavaScript frameworks or libraries.

Let's bootstrap the web application by creating a *motions.html* page, seen in Example 7-1.

Example 7-1. Template for the motions.html web application

```
<!DOCTYPE html>
<html>
<head>
  <meta content="width=device-width" name="viewport">
  <meta charset="utf-8">
  <title>Motions - MQTT Example</title>
  <link rel="stylesheet" type="text/css" href="http://bgrins.github.com
  /spectrum/spectrum.css">
</head>
<body>
  <h1>Motions - MQTT Example</h1>

  <h2>Devices</h2>
  <ul id="devices">
  </ul>

  <footer>&copy; 2014 <a href="http://mobile-web-messaging.net">Mobile &amp; Web
Messaging</a></footer>

  <script src='mqttws31.js'></script>
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js">
  </script>
  <script src="http://omnipotent.net/jquery.sparkline/2.1.2/
  jquery.sparkline.min.js"></script>
 <script>>
$(document).ready(function() {

// We will put all the JavaScript code in this block that is called
// when the document is ready

});
</script>
</body>
</html>
```

Create an MQTT Client with mqttws31.js

When the browser loads this page, we will create an MQTT client using the JavaScript library.

To create the client, we must pass the MQTT broker host and port (in our case, the host is *iot.eclipse.org* and the port is 80). Note that in "Create an MQTT Client with MQTTKit" on page 109, the Motions iOS application was connecting to *iot.eclipse.org* on its default port 1883. This default port expects a TCP socket connection. Because the web application uses HTML5 Web Sockets, it must connect to the 80 port that will handle the HTTP Upgrade process from the initial HTTP connection.

We also need to configure a `clientID` that identifies the MQTT client. In our case, we will just use a random string:

```
$(document).ready(function() {

  var host = "iot.eclipse.org";
  var port = 80;
  var clientID = Math.random().toString(12);

  var devices = {};

  var client = new Messaging.Client(host, Number(port), clientID);
}
```

Connect to the MQTT Broker

Once the `client` is created, we need to connect it to the MQTT broker. There are four steps to achieve this in a proper fashion:

1. Define an `onConnectionLost` handler that will be called back if the connection is lost *after the client has been successfully connected.*

2. Define an `onSuccess` handler that will be called if the client is successfully connected.
3. Define an `onFailure` handler that will be called if the client fails to connect to the broker.
4. Call `client.connect()` and pass the `onSuccess` and `onFailure` handlers in an object parameter.

Here's the code to accomplish this:

```
client.onConnectionLost = function(response) {
  if (response.errorCode !== 0) {
    alert(response.errorMessage + "\nclientID = " + client.clientID +
    " [" + response.errorCode + "]\n");
  }
};
client.connect({onSuccess: function(frame) {
    // this function is executed after a successful connection to
    // the MQTT broker
  },
  onFailure: function(failure) {
    alert(failure.errorMessage);
  }
});
```

All these handlers are optional and you may omit them. However, without them, the web application may fail to connect or lose connection without any way to let the user be aware of it.

Receive MQTT Messages

Once the client is successfully connected to the MQTT broker (i.e., when the `onSuccess` handler is called), we can then subscribe to a topic to receive all the device motions.

The MQTT client does not know the whole list of devices that send their motion data, so it cannot subscribe to specific MQTT topics.

Fortunately, MQTT defines wildcards for topics, which is useful for this case.

Topic Wildcards

Three characters have a special meaning when they are used in an MQTT topic:

Topic level separator `/`
: The forward slash (/) is used to separate each level within a topic tree and provide a hierarchical structure to the topic space. The use of the topic level separa-

tor is significant when the two wildcard characters are encountered in topics specified by subscribers.

Multilevel wildcard #
: The number sign (#) is a wildcard character that matches any number of levels within a topic.

Single-level wildcard +
: The plus sign (+) is a wildcard character that matches only one topic level.

The web application is interested in receiving any messages sent to topics of the form `/mwm/XXX/motion` where `XXX` is the device identifier. It maps to the MQTT wildcard topic `/mwm/+/motion`.

Note that it would not have been a good idea to use the more general wildcard `/mwm/#` (using the multilevel wildcard) as it would have matched both `/mwm/XXX/motion` *and* `/mwm/XXX/alert`. The web application is not interested in the alert sent to the devices. It is better to subscribe to the most specific wildcard topic instead of being too general and it filters out message later. This also preserves network bandwidth and CPU usage; the broker will not deliver messages to the client over the network and the client will not process them before they are discarded.

```
client.connect({onSuccess: function(frame) {
     // when the client is successfully connected,
     // subscribe to all the motions topics
    client.subscribe("/mwm/+/motion");
 },
    ...
  });
```

We have subscribed to the `/mwm/+/motion` wildcard topic, but how do we handle messages that will be delivered by the broker for all the topics that match?

The `client` object has an `onMessageArrived` property that will be called every time a message is delivered to the client. This property must be a function that takes a single `message` parameter corresponding to the MQTT message that is delivered to the client.

This `message` object defines several properties representing the MQTT message data. The `destinationName` property contains the actual name of the topic that delivered this message. Because we used meaningful topic names of the form `/mwm/XXX/motion`, we can extract the `deviceID` from the `destinationName`.

The `message` object defines two properties to receive its payload content:

- `payloadBytes` corresponds to a `ArrayBuffer` representation of the message payload.

- `payloadString` corresponds to a UTF-8 string representation of the message payload. This property can only be used if the payload is composed of valid UTF-8 characters.

In "Message representation for the Motions application" on page 11, we decided to send the device motions data as an array of 3 64-bit floats corresponding to the motions pitch, roll, and yaw values.

To be able to get these values, we must use the `payloadBytes` property and use a `DataView` to retrieve the three values for this array.

After we got these pitch, roll, and yaw values, we call the `updateSparklines()` method to update the sparkline for the given `deviceID`:

```
// subscription callback
client.onMessageArrived = function(message) {
  // get the device's ID from the message's destination
  var deviceID = message.destinationName.split("/")[2];

  // get the device data from the message payload as a byte array
  var data = message.payloadBytes;
  // use a DataView on the data buffer to get the three motions values as
  // double (aka Float64)
  var values = new DataView(data.buffer);
  var pitch = values.getFloat64(data.byteOffset);
  var roll = values.getFloat64(data.byteOffset +
  Float64Array.BYTES_PER_ELEMENT);
  var yaw = values.getFloat64(data.byteOffset + 2 *
  Float64Array.BYTES_PER_ELEMENT );

  updateSparklines(deviceID, pitch, roll, yaw);
};
```

Draw Sparklines

The `updateSparklines()` method will store the motions values in the `devices` object that was created when the page was loaded. It will create the HTML elements to display the data and use jQuery Sparklines to display them in a graphic.

The `devices` object is a map with keys that contain the `deviceID` of the devices that are sending the motion data. The values will be composed of three arrays to store the received value for pitch, roll, and yaw. We will only keep the 50 most recent values.

We will create three separate sparklines for:

- `pitch` (displayed in red)
- `roll` (diplayed in green)

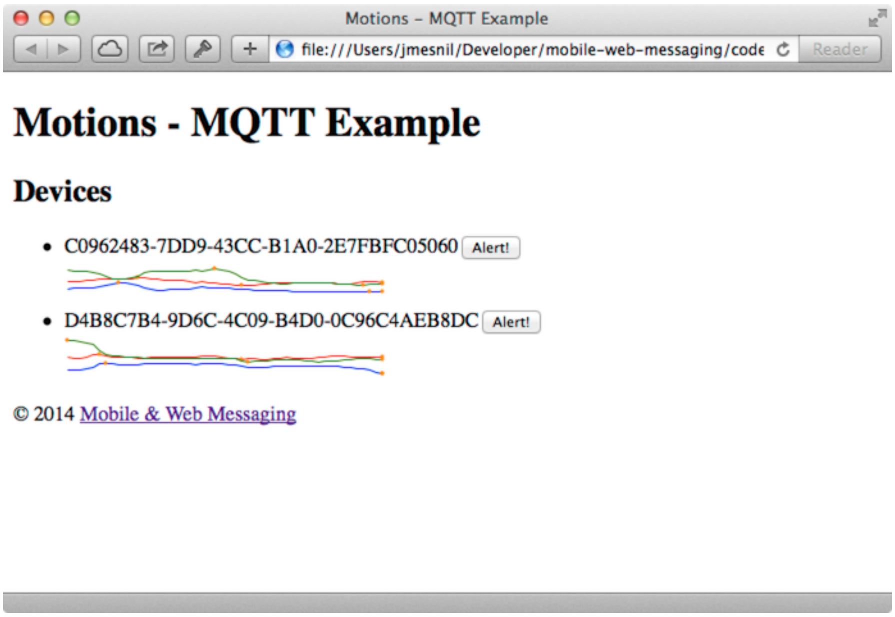
- `yaw` (displayed in blue)

These three sparklines will be composited in a single canvas that is drawn in the `<div class="data">` element created inside the `<div>` element identified by the `deviceID`:

```
function updateSparklines(deviceID, pitch, yaw, roll) {
  var values = devices[deviceID];
  // if the device is not known, create the UI for it
  if (!values) {
    var item = $('#devices').append(
      $('<li>').attr("id", deviceID).append(
        $('<label>').text(deviceID),
        $('<button>').text("Alert!").click(function() { sendAlert(deviceID); }),
        $('<br>'),
        $('<div>').attr('class', 'data')
      )
    );
    // create an empty array to hold its values
    values = {
      "pitch" : [],
      "roll" : [],
      "yaw" : [],
    };
  }
  // add the new value at the end of the array
  values.pitch.push(pitch);
  values.roll.push(roll);
  values.yaw.push(yaw);
  // keep only the 50 most recent values
  if (values.pitch.length > 50) {
    values.pitch.splice(0,1);
    values.roll.splice(0,1);
    values.yaw.splice(0,1);
  }
  // put back the updated values in the clients map
  devices[deviceID] = values;
  // display the values as a sparkline
  $('#'+ deviceID + ' .data').sparkline(values.pitch, {
    width: values.pitch.length * 5,
    tooltipPrefix: "pitch:",
    lineColor: 'red',
    fillColor: false,
    chartRangeMin: -3,
    chartRangeMax: 3,
    height: '36px'
  });
  $('#'+ deviceID + ' .data').sparkline(values.roll, {
    tooltipPrefix: "roll:",
    lineColor: 'green',
    composite: true,
    fillColor: false,
    chartRangeMin: -3,
    chartRangeMax: 3
   });
   $('#'+ deviceID + ' .data').sparkline(values.yaw, {
    tooltipPrefix: "yaw:",
    lineColor: 'blue',
    composite: true,
    fillColor: false,
    chartRangeMin: -3,
    chartRangeMax: 3
   });
  }
```

Note that we also create an "Alert!" button for each device that calls the `sendAlert()` method with the `deviceID` when the button is clicked. We will implement this method in the next section.

At this stage, we can now load the application in a web browser. If there are devices that are running the Motions iOS application, we will see them appear automatically on the page (Figure 7-2).



Motions - MQTT Example

Devices

- C0962483-7DD9-43CC-B1A0-2E7FBFC05060
- D4B8C7B4-9D6C-4C09-B4D0-0C96C4AEB8DC

© 2014 Mobile & Web Messaging

Figure 7-2. Two devices publishing their motions via the Motions application

Send MQTT Messages

We now have a web application that receives MQTT messages.

The other feature of this web application is to *send* an MQTT message to an alert topic so that the device that subscribes to this topic will change its background color using the message payload.

When the HTML elements for a device were created, we added a `<button>` that calls `sendAlert(deviceID)` when the user clicks on it.

In this method, we will create an MQTT message object using the `new Messaging.Message()` constructor and pass a `"red"` to it to set its payload.

The message object has a `destinationName` property that must be set prior to sending the message. We use the `deviceID` to build the name of the topic corresponding to this device alert: `"/mwm/" + deviceID + "/alert"`.

Finally, the last step is to call `client.send()` and pass it the `message` to send it to the topic. Note that the `client` was already connected when the page was loaded:

```
function sendAlert(deviceID) {
  // create a message with an empty payload
  var message = new Messaging.Message("red");
  message.destinationName = "/mwm/" + deviceID + "/alert";
  client.send(message);
}
```

If we reload the web application and click on an "Alert!" button, the corresponding device will receive the message from its alert topic and the code that we wrote in "Receive MQTT Messages" on page 116 will be executed to temporarily change the background color of the device (Figure 7-3).

Summary

In this chapter, we learned to use MQTT over Web Sockets to send and receive MQTT messages from a web application.

We use two different types of message payload:

- A binary payload composed of three 64-bit floats
- A UTF-8 string payload

To send a message, the application must do the following:

1. Connect to the MQTT broker.
2. Send the message to a topic.

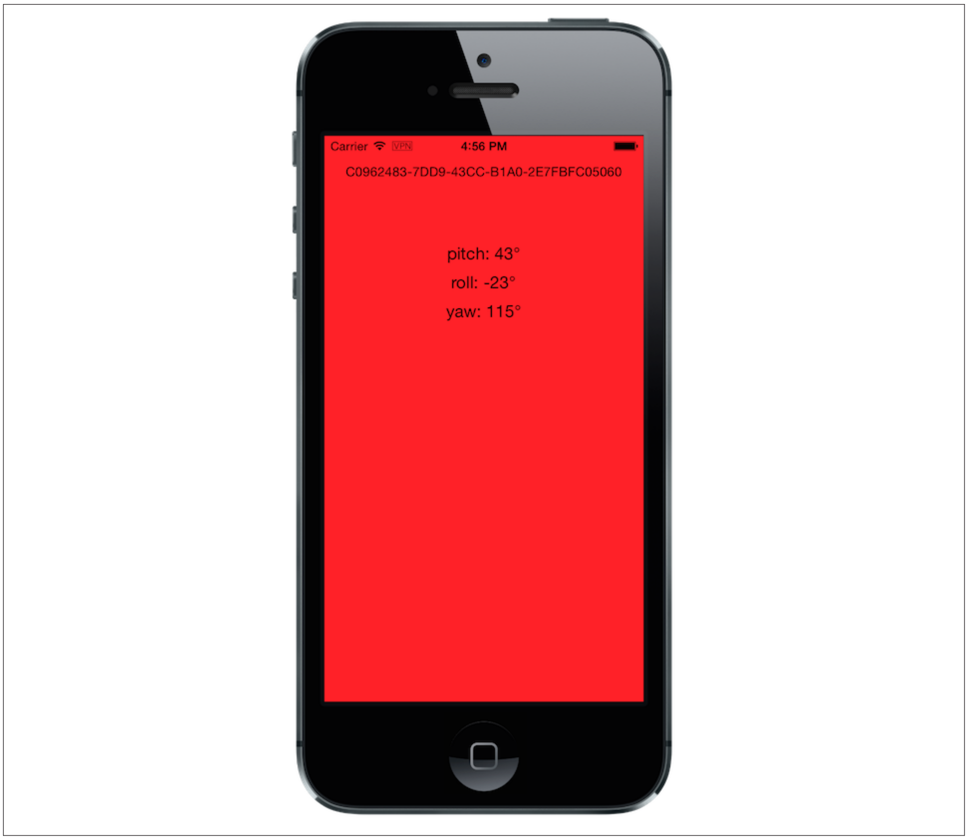


Figure 7-3. The Motions application background becomes red when an alert is received from its alert topic

To consume a message, the application must do the following:

1. Connect to the MQTT broker.
2. Subscribe to a (potentially wildcard) topic and set a handler that is called every time a message is received.

In the next chapter, we will learn about more advanced features of MQTT that were not required to write this simple application. However, it is likely that you may need some of these features if your applications are more complex.

CHAPTER 8

Advanced MQTT

In Chapters 6 and 7, we used MQTT to send and receive messages from a native iOS application and a web application. MQTT provides additional features that we did not use to write these applications. In this chapter, we will take a tour of all these advanced features provided by MQTT.

This chapter covers the latest version of the protocol at the time of this writing (i.e., MQTT v3.1, which was released on August 19, 2010).

Authentication

In Chapters 6 and 7, we connected to the Eclipse public MQTT broker that accepts *unauthenticated* connections. We did not need to pass any user credentials, as they would not be checked by the broker anyway.

If you are using an MQTT broker that is configured to accept secured connections, the client needs to pass a *username* and *password* when it connects to the broker.

MQTTKit Example

The `MQTTClient` has two `NSString` properties that must be set to authenticate the client, `username`, and `password`. They must be set *prior* to calling the client's `connect` methods in order to take effect.

If the MQTT broker requires authentication, the client can check if the connection was refused due to invalid user credentials using the `ConnectionRefusedBadUserNameOrPassword` error code from the `completionHandler`:

```
- (void)connect
{
    NSString *username = @"...";
```

```
        NSString *password = @"...";

        self.client.username = username;
        self.client.password = password;
        NSLog(@"Connecting to %@...", kMqttHost);
        [self.client connectToHost:kMqttHost
                 completionHandler:^(MQTTConnectionReturnCode code) {
            if (code == ConnectionAccepted) {
                NSLog(@"connected to the MQTT broker");
                [self subscribe];
            } else if (code == ConnectionRefusedBadUserNameOrPassword) {
                NSLog(@"Failed to authenticate the user");
            } else {
                NSLog(@"Failed to connect to the MQTT broker: code=%lu",
                (unsigned long)code);
            }
        }];
    }
```

mqttws31.js Example

To authenticate using the `mqttws31.js` library, you must set the `userName` and `pass word` properties to the object passed to the client's `connect` method.

If the MQTT broker requires authentication and the client passes invalid user credentials, the client will be notified by having its `onFailure` handler called with a failur code set to `4` (the value specified in the MQTT protocol for the error `Connection Re fused: bad user name or password`):

```
var userName = "...";
var password = "...";

client.connect({onSuccess: function(frame) {
    ...
  },
  onFailure: function(failure) {
   if (failure.code === 4) {
     alert("invalid user credentials");
     return;
   }
    ...
  },
  userName: userName;
  password: password;
});
```

Error Handling

The MQTT protocol does not specify any error handling. An MQTT broker has no possible way to inform a client that an error occured. The only action that the MQTT broker can take is to close the underlying network connection so that the client is no longer connected to the broker.

MQTT libraries provide callbacks or handlers for these cases so that they can act when such an error occurs.

MQTTKit Example

The `MQTTClient` class has a `disconnectionHandler` property that can be set to handle any unexpected error leading to a disconnection. The disconnection can be the consequence of the MQTT broker closing the network exception (in case of abnormal errors or an administrative operation) or the network connection can be directly broken (e.g., if a mobile device is no longer able to receive any signal).

The `disconnectionHandler` is a block that takes an `NSUinteger code` parameter. There are no standard values for the code and you will have to consult your MQTT broker documentation if you need to act differently depending on the type of errors.

We can modify the `Motions` iOS application to handle such disconnection failures and aggressively try to reconnect the MQTT broker (assuming the root cause of the disconnection failures are transient):

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    ...

    self.client = [[MQTTClient alloc] initWithClientId:self.deviceID];

    // use a weak reference to avoid a retain/release cycle in the block
    __weak MWMViewController *weakSelf = self;
    self.client.disconnectionHandler = ^(NSUInteger code) {
        NSLog(@"client disconnected with code %lu", (unsigned long)code);
        NSLog(@"trying to reconnect...");
        // trying to reconnect
        [weakSelf connect];
    };

    ...

    [self connect];
}
```

mqttws31.js Example

In "Connect to the MQTT Broker" on page 125, we already set an `onConnectionLost` callback on the `client` to be notified in case of a connection error. The callback has a single `response` parameter object. This parameter is composed of two properties:

- `errorCode`, a numerical representation of the type of connection error
- `errorMessage`, a textual description of the error

We can modify the `Motions` web application to display an alert message when a connection loss is detected to inform the user:

```
client.onConnectionLost = function(response) {
  alert(response.errorMessage + "\nclientID = " + client.clientID + " [" +
  response.errorCode + "]\n");
};
```

Heart-Beating

MQTT offers a simple mechanism to test the health of the network connection between a client and a broker using heart-beating.

Heart-beating is enabled by specifying a *keep alive timer* when the client initially connects to the broker. This timer, measured in seconds, defines the maximum time interval between messages received from a client. It allows the client and broker to detect whether the network connection is broken without waiting for the long TCP/IP timeout. A timer value of `0` disables heart-beating.

In the absence of regular messages exchanged between them, the client and the broker automatically send respective heart-beats (`PINGREQ` for the client and `PINGRESP` for the broker) based on the keep alive timer to check the health of the network connection.

If the client does not receive heartbeats from the broker, it will close the underlying network connection and report an error.

If the broker does not receive heartbeats from the client, it will consider the client to be disconnected.

Setting a good value for the keep alive timer is highly dependent on the application use cases and the platform it runs on.

For mobile devices that are subject to frequent intermittent network failures, using a value too small will report false failures and increase the instability of the application. It will also increase the bandwidth and battery usage as heartbeats would have to be sent over the network more frequently.

MQTTKit Example

By default, MQTTKit defines a keep-alive timer of 60 seconds.

It is possible to change this value using the keepAlive property on the MQTTClient object. The property type is a short and its value must be changed prior to calling the client's connect method in order to take effect:

```
MQTTClient *client = [[MQTTClient alloc] initWithClientId:clientID];
client.keepAlive = 10; // seconds
[client connectToHost:host
    completionHandler:^(MQTTConnectionReturnCode code) {
        //...
}];
```

mqttws31.js Example

mqttws31.js also defines a keep-alive timer of 60 seconds by default.

The client's connect method can take an optional keepAliveInterval integer to specify another value (or 0 to disable heart-beating):

```
client.connect({onSuccess: function(frame) {
    ...
  },
  onFailure: function(failure) {
    ...
  },
  keepAliveInterval: 10 // seconds
});
```

Last Will

One strength of messaging protocols is that producers and consumers are loosely coupled. They do not have to be online at the same time to exchange messages. The producer can send a message to a destination and be terminated. The messaging broker will then deliver the message to a consumer when it subscribes to this destination.

However, there are cases in which an application may require more information on the liveness of messaging clients.

Let's take the example of the Motions application that broadcasts the device position when it moves. A consumer of the device position topic will consume these messages. However, how could the consumer distinguish between receiving the messages because the device does not move or because the device is offline and has stopped broadcasting its position?

If the device is offline, the consumer may want to be notified to discard the device position from the map or show it differently from other *live* devices.

MQTT provides a *last will* feature that we could use to handle this use case.

When an MQTT client connects to the broker, it can specify a last will message that will be published to a last will topic by the broker *on behalf* of the client in case of unexpected disconnection. If the client disconnects normally, its last will message is not published. If the client uses heart-beating and the broker fails to receive its heart-beat in a timely fashion, this is considered an unexpected disconnection and the last will message will be published.

We could use this last will to let consumers know that the Motions iOS application has been terminated abnormally or its device is no longer reachable (in case of net-work disconnection).

MQTTKit Example

The `STOMPClient` object has `setWill:toTopic:withQos:retain` and `setWillData:toTopic:withQos:retain` methods to specify the client's last will. The difference between the two methods is that the first one takes an `NSString` for the will message payload and the second takes an `NSData`. These methods must be called before the client connects to the MQTT broker to take effect.

We could improve the `Motions` iOS application by specifying a last will to its `client` object in *MWMViewController.m* before it connects.

The last will topic can be any MQTT topic. We will use the `/mwm/lastWill` topic so that a consumer would have to subscribe to this topic to be notified of any device's abnormal disconnection. The payload of the last will message is a simple JSON object with a `deviceID` property. We will encapsulate the setup of the last will in a `setLastWill` method:

```
- (void)setLastWill
{
    NSString *willTopic = @"/mwm/lastWill";
    NSDictionary *dict = @{ @"deviceID": self.deviceID};
    NSData *willData = [NSJSONSerialization dataWithJSONObject:dict
                                                      options:0
                                                        error:nil];

    [self.client setWillData:willData
                     toTopic:willTopic
                     withQos:ExactlyOnce
                      retain:NO];
}
```

We just need to call this method before connecting to the MQTT broker in `connect`:

```
- (void)connect
{
    [self setLastWill];
    NSLog(@"Connecting to %@...", kMqttHost);
```

```
        [self.client connectToHost:kMqttHost
                 completionHandler:^(MQTTConnectionReturnCode code) {
            ...
        }];
    }
```

Similar to a regular message, the last will message can specify its QoS and whether it must be retained. A last will message might be important but infrequent. Using a QoS of `Exactly Once` will ensure that a consumer of the last will topic will not receive false positives on the device's disconnection. We will also not retain the last will message. If it were retained, a newly subscribed consumer could receive it and assume that a device has been disconnecting while it reconnected in the meantime.

Before we configure the web application's own last will, we can first update it to discard data when it receives the last will message from a device.

To achieve this, we need to do the following:

1. Subscribe to the last will topic `/mwm/lastWill`.
2. Update the subscription callback to handle last will messages.

The first step is done in the `onSuccess` callback passed to `clients` `connect` method when we were already subscribing to the devices, motion topics:

```
var lastWillTopic = "/mwm/lastWill";

client.connect({onSuccess: function(frame) {
  // after the client is successfully connected,
  // subscribe to all the motions topics
  client.subscribe("/mwm/+/motion");
  // subscribe to the last will topic, too
  client.subscribe(lastWillTopic);
},
```

The second step requires us to modify the `client`'s `onMessageArrived` callback to check whether the message is coming from the last will topic and discard the device data if that the case. Because the last will message representation is a JSON object, we must first parse it by calling `JSON.parse` on the message's `payloadString`:

```
client.onMessageArrived = function(message) {
  if (message.destinationName === lastWillTopic) {
    var payload = JSON.parse(message.payloadString);
    discard(payload.deviceID);
    return;
  }
  // the rest of the function is unchanged
  ...
};
```

The `discard` function will delete the data from the `device`'s dictionary and remove the HTML elements that were created to display the device:

```
function discard(deviceID) {
  console.log("discard data for " + deviceID);
  delete devices[deviceID];
  $('#'+ deviceID).remove();
}
```

mqttws31.js Example

It is also possible to set a client's last will using `mqttws31.js`. The `client`'s `connect` method can take an optional `willMessage` object that represents the last will message to send if it disconnects unexpectedly. The value is a regular MQTT message created by calling a `new Messaging.Message` constructor and specifying its `destinationName` (the last will topic), and optionally its `qos` and `retained` value:

```
var willMessage = new Messaging.Message("Web client " + clientID +
"has unexpectedly died");
willMessage.destinationName = "/mwm/lastWill/web";
willMessage.qos = 2; // exactly once
willMessage.retained = false;

// specify the last will when the client connects to the broker
client.connect({onSuccess: function(frame) {
    ...
  },
  onFailure: function(failure) {
    ...
  },
  willMessage: willMessage
});
```

Often, applications may not need to be notified of the last will of another MQTT client. However, we may still want to monitor the unexpected disconnection to be informed of the liveness of the whole system. If all MQTT clients have configured their last will, we can have a crude monitoring application by subscribing to their last will topics:

```
$ mosquitto_sub -h iot.eclipse.org -t /mwm/lastWill/# -v
...
/mwm/lastWill {"deviceID":"C0962483-7DD9-43CC-B1A0-2E7FBFC05060"}
/mwm/lastWill/web Web client 0.90778b769105b876 has unexpectedly died
```

We subscribed to the wildcard topic `/mwm/lastWill/#` to receive messages from both `/mwm/lastWill` (that is used by the Motions iOS application) and any of its children including `/mwmw/lastWill/web` (that is used by the Motions web application).

Clean Session

When an MQTT client connects to the broker, it can specify whether the broker must store its state after it disconnects and until it reconnects. The client state that is stored includes its subscriptions and any in-flight message with a QoS greater or equal to `1`. Messages with a QoS of `0` (`AtMostOnce`) are not stored, because they are delivered on a best effort basis.

The client uses a Clean Session flag for this. If the flag is set, the broker will not store any state and the connection opened by the client will be *clean*. If the flag is not set, the broker will store the client state.

A client with the Clean Session flag set will have to subscribe again to consume messages.

A client that does not set the Clean Session flag will consume memory on the broker side (to store its state) and the broker might also perform administrative operations to remove such state. Unless there is a strong incentive to use such a client, it is better practice to use a Clean Session client and subscribe again after it reconnects.

MQTTKit Example

By default, MQTT clients created using MQTTKit have the Clean Session flag set (their state is not stored by the broker after they disconnect). It is also possible to change this behavior by using the `MQTTClient`'s `initWithClientID:cleanSession:` initializer and passing `NO` to its `cleanSession` parameter:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    ...

    // do not clean the session in the broker when the client disconnects
    self.client = [[MQTTClient alloc] initWithClientID:self.deviceID
                                          cleanSession:NO];

    ...

    [self connect];
}
```

If the Motions iOS application is modified this way, we can test it by connecting to the broker (so that the broker knows that it must store its state) and closing the application.

While the application is closed, we will modify the *motions.html* web application to send an alert message to the device alert topic with a QoS of `1` (`At Least Once`):

```
function sendAlert(deviceID) {
  var message = new Messaging.Message("red");
  message.destinationName = "/mwm/" + deviceID + "/alert";
  // send the alert with a QoS of At Least Once
  message.qos = 1;
  client.send(message);
}
```

The client will not be available to receive the message so the broker must store it to deliver when the client reconnects.

If we open the `Motions` iOS application again, the broker will then deliver the message to the client.

mqttws31.js Example

The clients created by the mqttws31.js library also connect by default with the Clean Session flag set. It is possible to change this behavior by adding a `cleanSession` property set to `false` in the properties passed to the client's `connect` method:

```
// specify that the session must not be cleaned when the client connects to
// the broker
client.connect({onSuccess: function(frame) {
    ...
  },
  onFailure: function(failure) {
    ...
  },
  cleanSession: false
});
```

Beyond MQTT?

MQTT is a simple protocol well-suited to a limited set of applications that can be modeled using the publish/subscribe model.

Its small set of features makes it simple to understand and use, but it lacks some flexibility and you have to carefully consider whether it meets your requirements.

One missing feature of MQTT is the lack of headers in the message representation.

MQTT defines a fixed set of headers for its command messages (used to connect to the broker, send a message, create a subscription, etc.), but there is no general notion of a header that the application or the broker could add to the messages.

This impacts the expressiveness of the messages that are delivered using MQTT. As a simple example, the absence of headers means that there is no way to know what type of data to expect from the payload. HTTP and STOMP define a `content-type` header that can be queried to know the MIME type of the payload and extract it accordingly.

MQTT does not allow this and the consumer must have *a priori* knowledge of the payload type for a message before being able to read it.

The absence of headers also implies that the MQTT broker cannot provide additional features not covered by the protocol in an unobtrusive way. As we saw in Chapter 5, STOMP brokers can provide additional features such as persistence, priority, and expiration, and the client can use them by adding headers to the messages. There is no such mechanism for MQTT in its current version. There are some ongoing discussions to add application-specific headers in a future version of the protocol, but no agreement has been reached at the time of this writing.

If you require features that are not provided by the protocol (or another messaging model than *publish/subscribe*), you may lose all the benefits of using this simple protocol and write brittle code that puts all the complexity in your applications instead of relying on the broker's set of features. But if your requirements are fullfilled by MQTT features (as is the case for a lot of applications), you will be all set to use messaging in your applications.

Summary

MQTT is a simple protocol that provides few advanced features. However, these features can be handy to solve common issues encountered by messaging applications.

In this chapter, we learned to use:

- *Authentication* to ensure that only authenticated clients can communicate with the MQTT broker
- *Error handling* to face unexpected connection issues and eventually reconnect to the broker
- *Heart-beating* to ensure that the network connection between the client and broker is healthy (and to ensure that it will kill the connection if that is not the case)
- *Last will* to let the broker sent a message on behalf of the client in case of an unexpected disconnection
- Clean Session to preserve client state on the broker between connections

PART III
Appendixes

APPENDIX A

Apache ActiveMQ

In this appendix, we describe how to install and configure Apache ActiveMQ to support STOMP and MQTT protocols.

The Apache ActiveMQ is a popular and powerful open source messaging server. It is fast, and supports many cross-language clients and protocols. In this book, we will connect to it from the Objective-C and JavaScript languages using two messaging protocols: STOMP and MQTT.

Download and Installation

Apache ActiveMQ latest release at the time of writing this book is 5.9.0. It can be downloaded from its download page and the getting started page contains all the information to install it.

For our setup, ActiveMQ will be installed in the directory *~/mobilewebmsg/mybroker*.

After you have downloaded the archive for ActiveMQ, it can be installed and started by running the commands:

```
$ tar zxvf apache-activemq-5.9.0-bin.tar.gz
$ cd apache-activemq-5.9.0
$ mkdir ~/mobilewebmsg/
$ ./bin/activemq create ~/mobilewebmsg/mybroker
...
$ cd ~/mobilewebmsg/mybroker
$ ./bin/mybroker start
...
$ tail data/activemq.log
...
2014-06-11  15:17:17,113  |  INFO    |  Apache  ActiveMQ  5.9.0  (mybroker,
ID:jeff.local-50723-1402492636703-0:1) started |
org.apache.activemq.broker.BrokerService | main
...
```

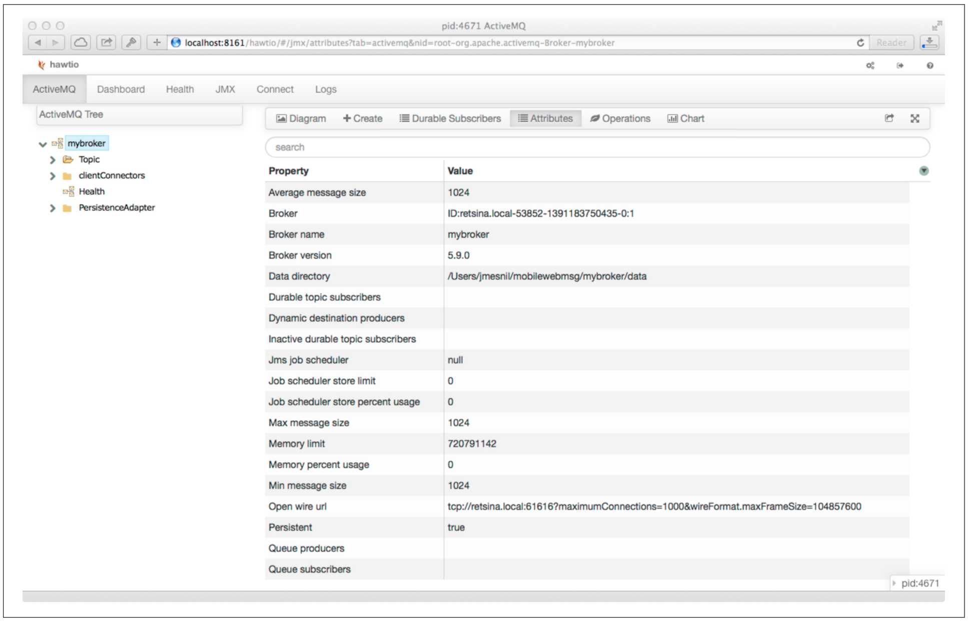
To stop the broker, run the following command:

```
    $ ./bin/mybroker stop
    ...
    Stopping broker: mybroker
    .... FINISHED
```

Administration Web Console

When the broker is started, it can be administered and managed using its web console running at *http://localhost:8161/hawtio*>. Out of the box, you can log into the admin console using the **`admin`** / **`admin`** credentials.

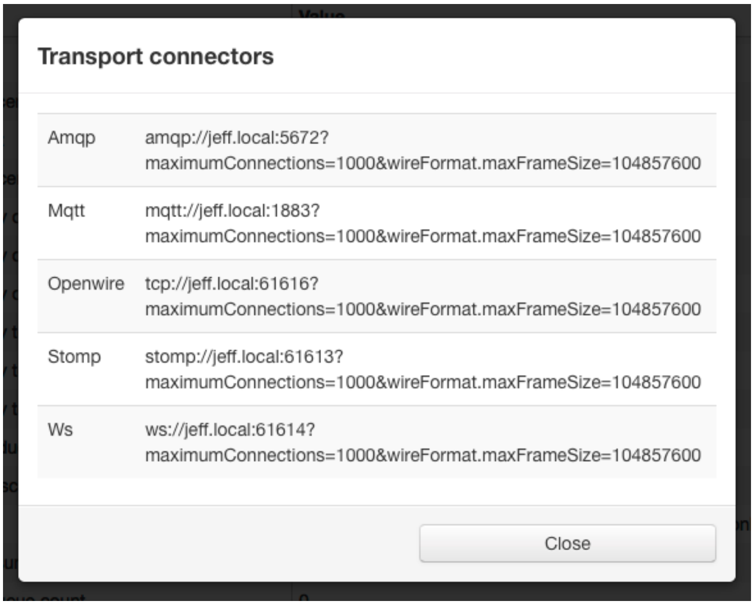
After you are logged in, you can check the `mybroker` resource in the ActiveMQ tree, as illustrated in Figure A-1.



Property	Value
Average message size	1024
Broker	ID:retsina.local-53852-1391183750435-0:1
Broker name	mybroker
Broker version	5.9.0
Data directory	/Users/jmesnil/mobilewebmsg/mybroker/data
Durable topic subscribers	
Dynamic destination producers	
Inactive durable topic subscribers	
Jms job scheduler	null
Job scheduler store limit	0
Job scheduler store percent usage	0
Max message size	1024
Memory limit	720791142
Memory percent usage	0
Min message size	1024
Open wire url	tcp://retsina.local:61616?maximumConnections=1000&wireFormat.maxFrameSize=104857600
Persistent	true
Queue producers	
Queue subscribers	

Figure A-1. The mybroker resource in the ActiveMQ web console

One interesting attribute is `transportConnectors`. If you click on the attribute value, a pop-up screen opens and display all the protocols supported by ActiveMQ out of the box (Figure A-2).


Transport connectors

Amqp	amqp://jeff.local:5672?maximumConnections=1000&wireFormat.maxFrameSize=104857600
Mqtt	mqtt://jeff.local:1883?maximumConnections=1000&wireFormat.maxFrameSize=104857600
Openwire	tcp://jeff.local:61616?maximumConnections=1000&wireFormat.maxFrameSize=104857600
Stomp	stomp://jeff.local:61613?maximumConnections=1000&wireFormat.maxFrameSize=104857600
Ws	ws://jeff.local:61614?maximumConnections=1000&wireFormat.maxFrameSize=104857600

Figure A-2. ActiveMQ default transport connectors

For this book, the important ones are:

- Stomp (`stomp://jeff.local:61613`)
- Mqtt (`mqtt://jeff.local:1883`)
- Ws (`ws://jeff.local:61614`)

They mean that to connect to ActiveMQ using the STOMP protocol we must connect to the `jeff.local` host (this is the name of my machine on my local network; your ActiveMQ installation will have another value) on the port `61613`.

Likewise, to connect to ActiveMQ using the MQTT protocol, we must again use the `jeff.local` host, but on the port `1883` this time.

Finally, there is also the Web Socket connector that must be used to connect from a web browser, *regardless* of the messaging protocol that we use. A web application would have to connect to the `61614` port regardless of whether the application is using STOMP or MQTT.

APPENDIX B

Mosquitto

In this appendix, we describe how to install the Mosquitto broker and its tools to produce and consume MQTT messages from the command line.

Download and Installation

Mosquitto is an open source message broker that implements the MQTT protocol. The public MQTT broker hosted on *iot.eclipse.org* that we used in Chapter 6 and Chapter 7 is an instance of Mosquitto.

Mosquitto also comes with two command-line tools to send and receive MQTT messages. These tools are handy when writing an application using MQTT to be able to quickly produce and consume messages and check that the application is working as expected.

To install Mosquitto on your operating system, please consult its documentation page, which contains instructions for various operating systems and package managers.

Produce Messages with mosquitto_pub

The `mosquitto_pub` command-line tool can be used to produce MQTT messages.

In this book, we use it in the simplest fashion to send a message. The parameters we must pass to the tool are:

- `-h host` to specify the host of the MQTT broker
- `-t topic` to specify the topic of the MQTT message
- `-m message` to specify the message payload to send

For example, to send a message with the text `yellow` for payload on the topic `/mwm/XYZ/alert` on the broker hosted on *iot.eclipse.org*, the command line is:

```
$ mosquitto_pub -h iot.eclipse.org -t /mwm/XYZ/alert -m yellow
```

The full list of the `mosquitto_pub` parameters is available by calling `mosquitto_pub --help`:

```
$ mosquitto_pub --help
mosquitto_pub is a simple mqtt client that will publish a message on a single
topic and exit.
mosquitto_pub version 1.3.1 running on libmosquitto 1.3.1.

Usage: mosquitto_pub [-h host] [-p port] [-q qos] [-r] {-f file | -l | -n | -m
message} -t topic
                     [-A bind_address] [-S]
                     [-i id] [-I id_prefix]
                     [-d] [--quiet]
                     [-M max_inflight]
                     [-u username [-P password]]
                     [--will-topic [--will-payload payload] [--will-qos qos]
                     [--will-retain]]
                     [{--cafile file | --capath dir} [--cert file] [--key file]
                      [--ciphers ciphers] [--insecure]]
                     [--psk hex-key --psk-identity identity [--ciphers ciphers]]
       mosquitto_pub --help

 -A : bind the outgoing socket to this host/ip address. Use to control which
interface the client communicates over.
 -d : enable debug messages.
 -f : send the contents of a file as the message.
 -h : mqtt host to connect to. Defaults to localhost.
 -i : id to use for this client. Defaults to mosquitto_pub_ appended with the
process id.
 -I : define the client id as id_prefix appended with the process id. Useful
for when the broker is using the clientid_prefixes option.
 -l : read messages from stdin, sending a separate message for each line.
 -m : message payload to send.
 -M : the maximum inflight messages for QoS 1/2..
 -n : send a null (zero length) message.
 -p : network port to connect to. Defaults to 1883.
 -q : quality of service level to use for all messages. Defaults to 0.
 -r : message should be retained.
 -s : read message from stdin, sending the entire input as a message.
 -S : use SRV lookups to determine which host to connect to.
 -t : mqtt topic to publish to.
 -u : provide a username (requires MQTT 3.1 broker)
 -P : provide a password (requires MQTT 3.1 broker)
 --help : display this message.
 --quiet : don't print error messages.
 --will-payload : payload for the client Will, which is sent by the broker in
 case of unexpected disconnection. If not given and will-topic is set, a zero
```

```
     length message will be sent.
     --will-qos : QoS level for the client Will.
     --will-retain : if given, make the client Will retained.
     --will-topic : the topic on which to publish the client Will.
     --cafile : path to a file containing trusted CA certificates to enable
     encrypted communication.
     --capath : path to a directory containing trusted CA certificates to enable
     encrypted communication.
     --cert : client certificate for authentication, if required by server.
     --key : client private key for authentication, if required by server.
     --ciphers : openssl compatible list of TLS ciphers to support.
     --tls-version : TLS protocol version, can be one of tlsv1.2 tlsv1.1 or tlsv1.
                     Defaults to tlsv1.2 if available.
     --insecure : do not check that the server certificate hostname matches the
     remote hostname. Using this option means that you cannot be sure that the
     remote host is the server you wish to connect to and so is insecure. Do not
     use this option in a production environment.
     --psk : pre-shared-key in hexadecimal (no leading 0x) to enable TLS-PSK mode.
     --psk-identity : client identity string for TLS-PSK mode.

    See http://mosquitto.org/ for more information.
```

Consume Messages with mosquitto_sub

The `mosquitto_sub` command-line tool can be used to consume MQTT messages.

The minimal parameters we must pass to the tool are:

- `-h host` to specify the host of the MQTT broker
- `-t topic` to specify the topic to consume from

For example, to consume messages from the topic `/mwm/XYZ/alert` on the broker hosted on *iot.eclipse.org*, the command line is:

```
$ mosquitto_sub -h iot.eclipse.org -t /mwm/XYZ/alert
...
yellow
```

Each message consumed by the tool will be displayed on a new line. You can type Ctrl+c to exit the tool and stop consuming messages.

The full list of the `mosquitto_sub` parameters is available by calling `mosquitto_sub --help`:

```
$ mosquitto_sub --help
mosquitto_sub is a simple mqtt client that will subscribe to a single topic and
print all messages it receives.
mosquitto_sub version 1.3.1 running on libmosquitto 1.3.1.

Usage: mosquitto_sub [-c] [-h host] [-k keepalive] [-p port] [-q qos] [-R] -t
topic ...
```

```
                     [-T filter_out]
                     [-A bind_address] [-S]
                     [-i id] [-I id_prefix]
                     [-d] [-N] [--quiet] [-v]
                     [-u username [-P password]]
                     [--will-topic [--will-payload payload] [--will-qos qos]
                     [--will-retain]]
                     [{--cafile file | --capath dir} [--cert file] [--key file]
                      [--ciphers ciphers] [--insecure]]
                     [--psk hex-key --psk-identity identity [--ciphers ciphers]]
       mosquitto_sub --help

 -A : bind the outgoing socket to this host/ip address. Use to control which
interface the client communicates over.
 -c : disable 'clean session' (store subscription and pending messages when cli-
ent disconnects).
 -d : enable debug messages.
 -h : mqtt host to connect to. Defaults to localhost.
 -i : id to use for this client. Defaults to mosquitto_sub_ appended with the
process id.
 -I : define the client id as id_prefix appended with the process id. Useful
for when the broker is using the clientid_prefixes option.
 -k : keep alive in seconds for this client. Defaults to 60.
 -N : do not add an end of line character when printing the payload.
 -p : network port to connect to. Defaults to 1883.
 -q : quality of service level to use for the subscription. Defaults to 0.
 -R : do not print stale messages (those with retain set).
 -S : use SRV lookups to determine which host to connect to.
 -t : mqtt topic to subscribe to. May be repeated multiple times.
 -u : provide a username (requires MQTT 3.1 broker)
 -v : print published messages verbosely.
 -P : provide a password (requires MQTT 3.1 broker)
 --help : display this message.
 --quiet : don't print error messages.
 --will-payload : payload for the client Will, which is sent by the broker in
case of unexpected disconnection. If not given and will-topic is set, a zero
 length message will be sent.
 --will-qos : QoS level for the client Will.
 --will-retain : if given, make the client Will retained.
 --will-topic : the topic on which to publish the client Will.
 --cafile : path to a file containing trusted CA certificates to enable
 encrypted certificate based communication.
 --capath : path to a directory containing trusted CA certificates to
 enable encrypted communication.
 --cert : client certificate for authentication, if required by server.
 --key : client private key for authentication, if required by server.
 --ciphers : openssl compatible list of TLS ciphers to support.
 --tls-version : TLS protocol version, can be one of tlsv1.2 tlsv1.1 or tlsv1.
                 Defaults to tlsv1.2 if available.
 --insecure : do not check that the server certificate hostname matches the
 remote hostname. Using this option means that you cannot be sure that the
 remote host is the server you wish to connect to and so is insecure.
```

```
              Do not use this option in a production environment.
 --psk : pre-shared-key in hexadecimal (no leading 0x) to enable TLS-PSK mode.
 --psk-identity : client identity string for TLS-PSK mode.

See http://mosquitto.org/ for more information.
```

Index

About the Author(s)

Jeff Mesnil is a software developer who has made significant contributions to many open source middleware projects. He currently holds a Senior Software Engineer position at Red Hat in its JBoss Middleware Division.

Colophon

The animal on the cover of *Mobile and Web Messaging* is a song thrush (*Turdus philomelos*), a small bird that lives throughout Europe and western Asia. Its scientific name comes from the Latin word for thrush, as well as the mythological Greek character Philomela, who was transformed into a songbird after her tongue was cut out.

The song thrush is about 8-9 inches long and typically weighs 2-4 ounces. It has brown plumage on its back, and a light belly with black spots. Despite being a smaller bird, it has one of the loudest bird calls. Male song thrushes, in particular, are capable of learning more than 100 different calls, including imitations of other birds and manmade objects like telephones.

Song thrushes are omnivorous; their diet is mainly made up of insects, worms, fruit, and berries. They also eat snails, and have the unique habit of tapping the shells against rocks to get to the meat inside. Known as "snail anvils," these stones and the remains of snails are often found in areas where thrushes live. While thrushes usually nest in forests with thick undergrowth, in more developed areas such as England, the birds are most often found in gardens.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from Wood's *Animate Creation*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.