Building Real-time Mobile Solutions with MQTT and IBM MessageSight

- Provides practical guidance to getting started quickly with MQTT and IBM MessageSight
- Builds a mobile application (PickMeUp) by using MQTT and IBM MessageSight
- Shows typical usage patterns and guidance to expand the solution

Bryan Boyd
Joel Gauci
Michael P Robertson
Nguyen Van Duy
Rahul Gupta
Vasfi Gucer
Vladimir Kislicins
Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (October 2014)

This edition applies to IBM MessageSight Version 1.1 and MQTT Version 3.1.
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>1. Overview of MQTT</th>
<th>2. Getting started with MQTT</th>
<th>3. Overview of IBM MessageSight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Notices</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trademarks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preface</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Now you can become a published author, too!</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comments welcome</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stay connected to IBM Redbooks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 1. Overview of MQTT</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1 Building a Smarter Planet world</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.1 The Internet of Things (IoT)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.2 Smarter Planet concept</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.3 Telemetry and the Internet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2 MQTT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.1 Benefits of the MQTT protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.2 Basic concepts of the MQTT protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.3 The OASIS MQTT Technical Committee</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.4 The Eclipse Paho project</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.5 Comparison of MQTT and HTTP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3 MessageSight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4 Benefits of using MQTT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5 Where to use MQTT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5.1 Connected car</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5.2 Connected city</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5.3 Connected home</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5.4 Connected consumers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 2. Getting started with MQTT</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1 MQTT concepts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1.1 MQTT messaging</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1.2 MQTT client programming</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2 MQTT server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2.1 IBM MessageSight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2.2 IBM WebSphere MQ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2.3 Really Small Message Broker and MQ Telemetry daemon for devices</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2.4 Mosquito</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3 MQTT clients</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.1 Eclipse Paho clients</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.2 IBM Mobile Messaging and M2M Client Pack MA9B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.3 Preparation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.4 Building the sample MQTT application</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.5 MQTT publisher and subscriber in Java</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.6 MQTT publisher and subscriber in JavaScript</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 3. Overview of IBM MessageSight</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1 Features of MessageSight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.1 MessageSight is a developer-friendly solution</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.2 Connections to MessageSight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2 Messaging patterns of MessageSight</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2.1 Fan out broadcast</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2.2 Fan in per device notification</td>
<td>51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2.3 Fan out per device notification</td>
<td>52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2.4 Fan out per device request-reply</td>
<td>53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2.5 Fan in per device request-reply</td>
<td>53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3 Install the MessageSight virtual appliance</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4 Overview of the MessageSight web UI</td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4.1 Connect to the MessageSight appliance</td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4.2 The MessageSight Home page</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4.3 Administrator actions using the MessageSight web UI</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5 Overview of the MessageSight CLI</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5.1 Connect to the MessageSight appliance</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5.2 Administrator actions using the MessageSight CLI</td>
<td>71</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6 Message hubs, endpoints, and policies</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.1 Endpoints</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.2 Message hubs</td>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.3 Connection policies</td>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.4 Messaging policies</td>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.5 Endpoints</td>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.6 The DemoHub message hub</td>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.7 Configuring your first message hub using the MessageSight web UI</td>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.8 Configuring a message hub using the MessageSight CLI</td>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.9 Use the MessageSight SSH to deploy message hub configuration</td>
<td>84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 4. Typical network topology, messaging patterns, and considerations</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1 Network topology</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1.1 The architecture</td>
<td>91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2 Messaging patterns</td>
<td>91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2.1 Fan out broadcast</td>
<td>92</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2.2 Fan in per device notification</td>
<td>93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2.3 Fan out per device notification</td>
<td>94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2.4 Fan in per device request reply</td>
<td>97</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2.5 Fan out per device request reply</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3 Messaging considerations</td>
<td>102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3.1 Quality of service</td>
<td>102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3.2 Message size</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3.3 Message order</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3.4 Topic namespace</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3.5 Retained message</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 5. IBM MessageSight and the key fob remote application</td>
<td>105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1 Overview of the key fob remote application</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1.1 Application overview</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1.2 Testing the key fob remote application</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2 MessageSight configurations for the key fob remote application</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2.1 MessageSight basic configuration</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3 Security capabilities of the MessageSight appliance</td>
<td>119</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3.1 Adding security controls to the key fob remote application</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3.2 Adding security at the transport level using SSL or TLS.</td>
<td>135</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 6. Overview of the PickMeUp application</td>
<td>147</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1 Company A scenario</td>
<td>148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1.1 Company A business problem</td>
<td>148</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
11.1.4 Configure the project build settings .................................................. 227
11.1.5 Run the application ........................................................................... 229
11.1.6 Run PickMeUp for your iOS project ................................................... 230
11.2 Set up a PickMeUp Android project ....................................................... 230
  11.2.1 Prerequisites .................................................................................. 230
  11.2.2 Register with Google Maps API ....................................................... 231
  11.2.3 Android SDK Packages ................................................................... 231
  11.2.4 Run PickMeUp for your Android project .......................................... 231
11.3 Set up the PickMeUp back end ............................................................... 231
  11.3.1 Prerequisites .................................................................................. 231
  11.3.2 Register with Bluemix ..................................................................... 232
  11.3.3 Download, deploy, and run PickMeUp .............................................. 232
11.4 Set up the PickMeUp HTML5 project .................................................... 233
Appendix A. The MQTT protocol ................................................................. 235
  Quality of service (QoS) levels and flow .................................................... 236
    QoS Level 0: At most once delivery ......................................................... 236
    QoS Level 1: At least once delivery ......................................................... 236
    QoS Level 2: Exactly once delivery ......................................................... 237
    Assumptions of QoS levels 1 and 2 ......................................................... 238
  QoS determination ..................................................................................... 238
  Impact of QoS level on performance ......................................................... 239
  The MQTT client identifier ..................................................................... 239
  MQTT durable and non-durable subscribers ............................................ 240
  MQTT persistence .................................................................................... 240
    Message delivery from JMS to MQTT .................................................... 241
  The MQTT header ................................................................................... 241
  The MQTT keep alive timer .................................................................... 242
  Delivery of the MQTT retry message ...................................................... 243
  The MQTT last will and testament .......................................................... 243
  The MQTT retained flag on messages ..................................................... 244
  The TCP/IP stack .................................................................................... 244
Appendix B. Additional material ................................................................. 245
  Locating the web material ....................................................................... 245
  Using the web material .......................................................................... 245
    System requirements for downloading the web material ...................... 246
    Downloading and extracting the web material ..................................... 246
Related publications .................................................................................. 247
  IBM Redbooks ....................................................................................... 247
  Online resources .................................................................................... 247
  Help from IBM ....................................................................................... 249
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.
Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

- Bluemix™
- DataPower®
- developerWorks®
- Global Technology Services®
- IBM®
- InfoSphere®
- Redbooks®
- Smarter Planet®
- WebSphere®
- Worklight®

The following terms are trademarks of other companies:

Worklight is trademark or registered trademark of Worklight, an IBM Company.

ITIL is a registered trademark, and a registered community trademark of The Minister for the Cabinet Office, and is registered in the U.S. Patent and Trademark Office.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.
Preface

MQTT is a messaging protocol designed for the Internet of Things (IoT). It is lightweight enough to be supported by the smallest devices, yet robust enough to ensure that important messages get to their destinations every time. With MQTT devices, such as energy meters, cars, trains, mobile phones and tablets, and personal health care devices, devices can communicate with each other and with other systems or applications.

IBM® MessageSight is a messaging appliance designed to handle the scale and security of a robust IoT solution. MessageSight allows you to easily secure connections, configure policies for messaging, and scale to up to a million concurrently connected devices.

This IBM Redbooks® publication introduces MQTT and MessageSight through a simple key fob remote MQTT application. It then dives into the architecture and development of a robust, cross-platform Ride Share and Taxi solution (PickMeUp) with real-time voice, GPS location sharing, and chat among a variety of mobile platforms. The publication also includes an addendum describing use cases in a variety of other domains, with sample messaging topology and suggestions for design.

Authors

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Austin Center.

Bryan Boyd is a Solutions Software Engineer for IBM MessageSight. His primary focus is to rapidly prototype solutions with MQTT, MessageSight, and analytics technologies in emerging industries. He owns and maintains http://m2m.demos.ibm.com, a demo collection that showcases dynamic HTML5 applications using IBM MessageSight and MQTT for real-time analytics, communication, and collaboration. Bryan has delivered technical presentations about MQTT and application development at industry conferences and developer events. Bryan holds a Master’s Degree in Computer Science from Texas A&M University.

Joel Gauci is a Certified IT Specialist in the IBM WebSphere® Software group in France. Since 2006, Joel has been working for leading European firms on projects including IBM DataPower®, MessageSight, and API Management in various sectors. As a Client Technical Professional, Joel mainly works on MessageSight and API Management selling opportunities. He assists potential customers from basic presentation to complex architecture definition in the Internet of Things domain. Joel has authored several IBM Redbooks publications and articles related to DataPower appliances. Joel holds a Master’s Degree in Computer Science and a Master's Degree in Mechanics from the University Paris 6 in France.
Michael P Robertson is a Software Developer for IBM MessageSight. His main focus has been on testing the client side of MessageSight with the Java Message Service (JMS) and MQTT protocols. In addition to MessageSight, Michael contributes to the development of the GoLang and Objective-C MQTT Client libraries developed by the Eclipse Paho project. He also works on developing demo applications for MessageSight and MQTT.

Nguyen Van Duy is an Advisory IT Architect with IBM Global Technology Services® in Vietnam. He is an IBM Associate Certified IT Architect with solid experience in IBM and open technologies. On his current assignment, Duy works as the Technical Leader for the IBM Global Procurement Services Group in Vietnam to provide enterprise software development services. He is focusing on mobile solutions, including the creation of mobile solutions for IBM employees, and providing his expertise in assisting IBM customers with enterprise mobile engagements. His core experiences are in web, security, distributed computing models, and mobile technologies.

Rahul Gupta is an Advisory IT Architect with IBM Global Technology Services (GTS) in the US. He is a Certified Service-Oriented Architecture (SOA) Architect with nine years of professional experience in IBM messaging technologies. At his current assignment, he works as a middleware consultant for various clients in North America. His core experiences are in lab testing, performance tuning, and Level 3 development for IBM Integration Bus and WebSphere MQ. Rahul has been a technical speaker for messaging-related topics at various WebSphere conferences. He is a recognized inventor by the IBM innovation community.

Vasfi Gucer is an IBM Redbooks Project Leader with the IBM International Technical Support Organization. He has more than 18 years of experience in the areas of systems management, networking hardware, and software. He writes extensively and teaches IBM classes worldwide about IBM products. His focus has been on cloud computing for the last three years. Vasfi is also an IBM Certified Senior IT Specialist, Project Management Professional (PMP), IT Infrastructure Library (ITIL) V2 Manager, and ITIL V3 Expert.

Vladimir Kislicins works as a Mobile Developer and Consultant for ISSW Mobile Practice, Europe. His main focus is on the rapid prototype development and proof of concept projects. With a passion for mobile and web technologies, Vladimir has extensive experience with Hybrid mobile, as well as native Android development. Vladimir is co-author of several Prior Art publications that focus on optimizing software processes to reduce battery consumption on mobile devices.
Thanks to the following people for their contributions to this project:

Karen Wallace
International Technical Support Organization, Raleigh Center

Thanks to the authors of the previous editions of this book:

- Authors of *Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry*, SG24-8054:
  - Valerie Lampkin
  - Weng Tat Leong
  - Leonardo Olivera
  - Sweta Rawat
  - Nagesh Subrahmanym
  - Rong Xiang
  - Martin Keen

**Now you can become a published author, too!**

Here’s an opportunity to spotlight your skills, grow your career, and become a published author - all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at: [ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)

**Comments welcome**

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at: [ibm.com/redbooks](http://ibm.com/redbooks)
- Send your comments in an email to: redbooks@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400
Stay connected to IBM Redbooks

- Find us on Facebook:
  http://www.facebook.com/IBMRedbooks
- Follow us on Twitter:
  http://twitter.com/ibmredbooks
- Look for us on LinkedIn:
  http://www.linkedin.com/groups?home=&gid=2130806
- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:
- Stay current on recent Redbooks publications with RSS Feeds:
  http://www.redbooks.ibm.com/rss.html
Overview of MQTT

This chapter introduces the MQTT protocol and how it can be used to connect various types of smart devices and applications that are measuring, monitoring, and, in certain cases, controlling the world today. This chapter also introduces IBM MessageSight. MessageSight is an appliance-based messaging server that is designed to handle large numbers of connected clients and devices and process high volumes of messages with consistent latency.

This chapter includes the following topics:

- 1.1, “Building a Smarter Planet world” on page 2
- 1.2, “MQTT” on page 6
- 1.3, “MessageSight” on page 12
- 1.4, “Benefits of using MQTT” on page 15
- 1.5, “Where to use MQTT” on page 17
1.1 Building a Smarter Planet world

The concept of connecting our superabundance of devices to the Internet, known as the Internet of Things\(^1\) (IoT), is a critical foundation on which the IBM Smarter Planet\(^{®}\) vision will be realized. In addition, supporting the IoT are new, more advanced approaches to telemetry that make it possible to connect all kinds of devices, wherever they might be, to each other, to the Internet, and to the business enterprise.

One of these advancements is the MQTT messaging protocol (http://www.mqtt.org). This protocol is so lightweight that it can be supported by some of the smallest measuring and monitoring devices, and it can transmit data over far-reaching, sometimes intermittent networks. MQTT is a publish/subscribe messaging transport protocol that is optimized to connect physical world devices and events with enterprise servers and other consumers. MQTT is designed to overcome the challenges of connecting the rapidly expanding physical world of sensors, actuators, phones, and tablets, with established software processing technologies.

Before getting into the details of MQTT, let’s take a brief look at the evolving world that developers who are using MQTT are working to connect.

**Note:** Starting with Version 3.1.1, MQTT does not stand for Message Queue Telemetry Transport anymore.

1.1.1 The Internet of Things (IoT)

Anyone who has used a web browser and a search engine or social media site knows the power of the Internet to connect people to information or to other people. Yet, with the rise of various smart devices, the Internet will evolve to become the IoT (Figure 1-1), in which billions of interconnected smart devices are measuring, moving, and acting upon, sometimes independently, all the bits of data that make up daily life.

To imagine what the IoT might bring about in the next 10 or 20 years, think about the remarkable things we already can do remotely:

- A connected car can predict parts failure and schedule maintenance. Driving habits can be captured and used for calculating insurance.
- A doctor can examine a patient in a distant city and see real-time health status information, such as blood pressure, heart rate, and so on.
- An energy company can monitor windmills in the middle of the ocean and remotely diagnose and cut off the problematic units.
- A homeowner can view his home on a web page, complete with the status of interior devices, such as the security alarm, heating system, and more.

The IoT will go beyond connecting people to information and to other people. Devices are interacting with devices, creating what might eventually become a global central nervous system (Figure 1-2).
1.1.2 Smarter Planet concept

The IBM Smarter Planet concept is built on a set of pillars called the *Three I’s*, as listed here and illustrated in Figure 1-3 on page 5:

- **Instrumented**: Information is captured wherever it exists, such as through the use of remote sensors.
- **Interconnected**: Information is moved from the collection point to wherever it can be usefully consumed.
- **Intelligent**: Information is processed, analyzed, and acted upon to derive maximum value and knowledge.
The world is already increasingly instrumented, with examples ranging from tiny sensors and radio-frequency identification (RFID) tags in stand-alone products, through smartphones and location-aware global positioning system (GPS) devices to notebook PCs and embedded systems. These devices typically have enough computing power to gather and transmit data, and certain devices have enough to respond to requests to modify their behavior.

These devices also are nearly all connected to some extent. Most have, or will have, an Internet address with which they can communicate directly across local networks or indirectly by way of the cloud. So, the concept of the IoT has already begun to emerge.

The next steps, then, are gathering all of the data that is collected by these small, medium, or even large devices, routing that data to where it is best interpreted, and using the world’s vast computational resources to understand what is happening, and to respond as necessary to improve the quality of life.

1.1.3 Telemetry and the Internet

Telemetry technology allows things to be measured or monitored from a distance. In addition, today, improvements in telemetry technology make it possible to interconnect measuring and monitoring devices at different locations, and to reduce the cost of building applications that can run on these smart devices to make them even more useful.

People, businesses, and governments are increasingly turning to smart devices and telemetry to interact more intelligently with the world. A man shopping for groceries wants to know what is currently in his pantry at home. A woman heading to Austin, Texas, wants to know whether flights into that city are currently delayed by weather. A motorist driving across town wants to know whether the main highway is still blocked by the car crash that was reported on the news. A doctor with a patient due to arrive in the office at 3 p.m. wants to know, in the morning, whether the patient’s blood pressure is stable enough to make the trip safely.

Information to help with each of these decisions can be made available from a variety of smart meters and devices.
The challenge is in getting the information from the devices to the people and applications that want to use the information, and to do so in time for the information to be used effectively. Ideally, this will have the added ability for the users and applications to reply to the devices with new instructions or requests. If the devices are widely distributed geographically, or if they have limited storage or computational abilities, the challenges increase considerably, as do costs.

Fortunately, these challenges are being overcome through the use of improved telemetry technologies and communication protocols that are making it possible to send and receive this information reliably over the Internet, even if the network is unsteady or the monitoring device has little processing power.

MQTT provides telemetry technology to meet the information challenges of today's Internet users.

### 1.2 MQTT

MQTT is an extremely simple and lightweight messaging protocol. Its publish/subscribe architecture is designed to be open and easy to implement, with up to thousands of remote clients capable of being supported by a single server. These characteristics make MQTT ideal for use in constrained environments where network bandwidth is low or where there is high latency, and with remote devices that might have limited processing capabilities and memory.

#### 1.2.1 Benefits of the MQTT protocol

The MQTT protocol offers the following benefits:

- Extends connectivity beyond enterprise boundaries to smart devices
- Offers connectivity options that are optimized for sensors and remote devices
- Delivers relevant data to any intelligent, decision-making asset that can use it
- Enables massive scalability of deployment and management of solutions

MQTT minimizes network bandwidth and device resource requirements and also attempts to ensure reliability and delivery. This approach makes the MQTT protocol particularly well-suited for connecting machine-to-machine (M2M), which is a critical aspect of the emerging concept of the IoT.

The MQTT protocol includes the following highlights:

- Open and royalty-free for easy adoption.

  MQTT is open and standardized by the OASIS Technical Committee\(^2\), and MQTT connects to MessageSight, inside an enterprise network. This makes it easy to adopt and adapt for the wide variety of devices, platforms, and operating systems inside an enterprise network. This is depicted graphically in Figure 3-1 on page 48.

- A publish/subscribe messaging model that facilitates one-to-many distribution.

  The sending application or device does not need to know the specifics of the receiver, even its IP address.

\(^2\) OASIS: Advancing open standards for the information society; for details, visit this website: [https://www.oasis-open.org/org](https://www.oasis-open.org/org)
Ideal for constrained networks (low bandwidth, high latency, data limits, and fragile connections).

MQTT message headers are kept as small as possible. The fixed header is just two bytes, and its on-demand, push-style message distribution keeps network utilization low.

Multiple service levels allow flexibility in handling different types of messages.
Developers can designate that messages will be delivered at most once, at least once, or exactly once.

Designed specifically for remote devices with little memory or processing power.
Minimal headers, a small client footprint, and limited reliance on libraries make MQTT ideal for constrained devices.

Easy to use and implement with a simple set of command messages.
Many applications of MQTT can be accomplished using only CONNECT, PUBLISH, SUBSCRIBE, and DISCONNECT control packets.

Built-in support for loss of contact between client and server.
The server is informed when a client connection breaks abnormally, allowing the message to be re-sent or preserved for later delivery.

1.2.2 Basic concepts of the MQTT protocol

The MQTT protocol is built upon several basic concepts, all aimed at ensuring message delivery and keeping the messages as lightweight as possible.

Publish/subscribe
The MQTT protocol is based on the principle of publishing messages and subscribing to topics, which is typically referred to as a publish/subscribe model. In this model, clients can subscribe to topics that pertain to them and therefore receive messages that are published about those topics. Alternatively, clients can publish messages to topics, therefore making them available to all subscribers to those topics.

Topics and subscriptions
Messages in MQTT are published to topics, which can be thought of as subject areas. Clients, in turn, sign up to receive particular messages by subscribing to a topic. Subscriptions can be explicit, which limits the messages that are received to the specific topic, or they can use multi-level wildcard designators (#) or a single-level wildcard designator (+) to receive messages for a variety of related topics.

Quality of service (QoS) levels
MQTT defines three quality of service (QoS) levels for message delivery, with each level designating a higher level of effort by the server to ensure that the message gets delivered. Higher QoS levels ensure more reliable message delivery but might consume more network bandwidth or subject the message to delays due to issues, such as latency.

Retained messages
With MQTT, the server keeps the message even after sending it to all current subscribers. If a new subscription is submitted for the same topic, any retained messages are then sent to the new subscribing client.
Clean sessions and durable connections
When an MQTT client connects to the server, it sets the clean session flag. If the flag is set to true, all of the client’s subscriptions are removed when it disconnects from the server. If the flag is set to false, the connection is treated as durable, and the client’s subscriptions remain in effect after any disconnection. In this event, subsequent messages that arrive carrying a high QoS designation are stored for delivery after the connection is reestablished. Using the clean session flag is optional.

The client will (or message)
When a client connects to a server, it can inform the server that it has a will, or a message, that will be published to a specific topic or topics in the event of an unexpected disconnection. A will is particularly useful in alarm or security settings where system managers must know immediately when a remote sensor has lost contact with the network.

1.2.3 The OASIS MQTT Technical Committee
The purpose of the MQTT Technical Committee is to define an open publish/subscribe protocol for telemetry messaging that is designed to be open, simple, lightweight, and suited for use in constrained networks and multi-platform environments. The Technical Committee will accomplish this purpose through the refinement of an input specification.

Background and opportunity
Many industries are seeing a trend and a demand for solutions that capture physical world events for use by enterprise and web applications. There is a need to capture and integrate data that is captured from sensors, actuators, and other types of devices with a wide range of application middleware and web programming models. These devices can be in an enterprise, or in stores, cars, mobile phones, and more, to capture events and transmit them to back-end applications.

Needs and requirements
A simple, lightweight, and easy to implement messaging protocol is required to connect applications on embedded and mobile devices with servers used in web, enterprise, and other applications, where a lightweight messaging protocol is a necessity. The protocol needs to cope with the network, hardware, and challenges with other resources, yet still provide the QoS required by the application.

Experience with messaging and events across many industry domains has identified key requirements for such a messaging protocol:

► The protocol needs to provide bidirectional messaging support to uniformly handle both signals and commands. Additionally, the protocol is required to deliver messages to and from constrained devices that are connected over networks that have limited bandwidth. Basic QoS levels are needed to reflect trade-offs among bandwidth, availability, and delivery guarantees. A subscriber must be able to set up a QoS that works in conjunction with the available constraints.

► Connectivity cognizance to support intermittently connected networks and devices. The publish/subscribe infrastructure needs to provide message subscribers and, if necessary, a means for making a decision about likely connected, disconnected, and error states of the end devices in the network.

► Loose coupling is required to support dynamic system environments, where a heavy influx of messages and events needs to be made available to enterprise applications and other consumers in an unanticipated manner. Time, space, and synchronization decoupling is needed.
For constrained operations, the protocol has a means of supporting platforms, technologies, and networks that are driven by diverse equations of cost, technology, and physical constraints.

- Scalability that is suitable for supporting environments in which multiple devices, applications, and sensors need to be connected to a server infrastructure.

### The value of standardization

Connectivity solutions currently exist to integrate these types of systems. However, the paucity of standardized messaging protocols that are designed explicitly to address the needs and requirements listed in “Needs and requirements” on page 8 has become a hindrance in growing opportunities for the IoT. The standardization of an open protocol that conforms to these technical and market requirements can overcome the inhibitors by providing the following benefits:

- **Choices**: The standard protocol needs to provide implementation choices for the various devices, networks, and suppliers, with no limit of choices and adaptability over time.

- **Flexible Integration**: Standardization needs to enable integration between the plethora of constrained devices and enterprise applications. This integration will assist in the widespread adoption of the protocol.

- **Time to Market**: The protocol needs to be open and one that scales well from critical, embedded systems up to high volume enterprise transaction processing, and one that is data, platform, and language independent. This will shorten the time to market and support new levels of integration.

- **Skills**: A standard, based on protocol and programming models, will stimulate skilled developers and encourage the adoption and use of the protocol.

For more details about the OASIS MQTT Technical Committee, visit this website:

https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt

### 1.2.4 The Eclipse Paho project

The Eclipse Paho project\(^3\) sponsored by iot.eclipse.org\(^4\), was developed to provide scalable, open source implementations of open and standard messaging protocols that support the requirements of M2M integration with web and enterprise middleware and applications. This includes client implementations for use on embedded platforms, with corresponding server support as determined by the community.

One of the major objectives of the Eclipse Paho project is to provide an effective level of decoupling between the device and applications. The project initially started with MQTT publish/subscribe client implementations for use on embedded platforms, but in the future, it will bring corresponding server support as determined by the community.

For M2M devices and client developers to integrate, develop, and test messaging components end-to-end, the Eclipse Paho project provides the framework to support the testing and development of end-to-end device connectivity with a server. The MQTT application described in this chapter uses the Java client implementation from the Eclipse Paho project.

---

\(^3\) [The Paho project](http://www.eclipse.org/paho/)

\(^4\) IoT.eclipse.org is making the Internet of Things (IoT) development simpler: [http://iot.eclipse.org/](http://iot.eclipse.org/)
The Eclipse Paho 1.0 release provides client libraries, utilities, and test material for the MQTT and MQTT-SN messaging protocols. MQTT and MQTT-SN are designed for existing, new, and emerging solutions for M2M and IoT. The Eclipse Paho project includes client libraries in Java, C/C++, Python, and JavaScript for desktop, embedded, and mobile devices.

For more details about the Eclipse Paho project, see this website:
http://www.eclipse.org/paho

Information is also available about the MQTT V3.1 Java and C clients, which IBM contributed, and which you can download from the following web pages:
http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.java.git/
http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.c.git/

1.2.5 Comparison of MQTT and HTTP

Although comparison is often made between MQTT and other common protocols, the most useful comparison is with the hypertext transfer protocol (HTTP) for the following reasons:

► HTTP is the most widely used and available protocol. Almost all computing devices with a TCP/IP stack have it. In addition, because HTTP and MQTT are both based on TCP/IP, developers need to choose between them.

► The HTTP protocol uses a request-and-response model, which is currently the most common message exchange protocol. MQTT uses a publish/subscribe model. Developers need to understand the relative advantages of each type of model.

Quick comparison of MQTT and HTTP

Table 1-1 provides a quick comparison to help developers choose the most suitable messaging protocol for applications.

<table>
<thead>
<tr>
<th>Design orientation</th>
<th>MQTT</th>
<th>HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>Data centric</td>
<td>Document centric</td>
</tr>
<tr>
<td>Complexity</td>
<td>Simple</td>
<td>More complex</td>
</tr>
<tr>
<td>Message size</td>
<td>Small, with a compact binary header that is just two bytes in size</td>
<td>Larger, partly because status detail is text-based</td>
</tr>
<tr>
<td>Service levels</td>
<td>Three QoS settings</td>
<td>All messages get the same level of service</td>
</tr>
<tr>
<td>Extra libraries</td>
<td>Libraries for C (30 KB) and Java (100 KB)</td>
<td>Depends on the application (JavaScript Object Notation (JSON) or Extensible Markup Language (XML), but typically not small</td>
</tr>
<tr>
<td>Data distribution</td>
<td>Supports 1 to zero, 1 to 1, and 1 to ( n )</td>
<td>1 to 1 only</td>
</tr>
</tbody>
</table>
Detailed comparison of MQTT and HTTP

The following list is a fuller explanation of the critical differences between the MQTT and HTTP protocols when used for devices:

- **Design orientation:**
  - MQTT is data-centric. It transfers data content as a byte array. It does not care about content.
  - HTTP is document-centric. It supports the Multipurpose Internet Mail Extensions (MIME) standard to define content type, but constrained devices usually do not need this advanced feature.

- **Messaging pattern:**
  - MQTT uses a publish/subscribe messaging pattern that has loose coupling. Clients do not need to be aware of the existence of other devices. They just need to care about the content to be delivered or received.
  - HTTP uses a request/response messaging model. It is a basic and powerful messaging exchange pattern, but the client needs to know the address of all devices to which it connects.

- **Complexity of protocol:**
  - The MQTT specification is short. It has few message types and only the CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE, and DISCONNECT types are important for developers.
  - HTTP is a more complex protocol, with a specification that is more than 160 pages long. It uses many return codes and methods, such as POST, PUT, GET, DELETE, HEAD, TRACE, and CONNECT. It works well for hypermedia information systems, but constrained devices typically do not need all of its features.

- **Message size:**
  - MQTT is designed specifically for constrained devices. It includes only the features that are necessary to support them. The message header in MQTT is short, and the smallest packet size for a message is 2 bytes.
  - HTTP uses a text format, not a binary format, which allows for lengthy headers and messages. The text format is readable by human beings. Therefore, the HTTP protocol is easy to troubleshoot, which has contributed to its popularity. However, this format is more than is needed, or desirable, for constrained devices with limited computational resources in low-bandwidth network environments.

- **QoS levels:**
  - MQTT supports three QoS levels in message publication. Developers do not have to implement additional, complex features to ensure message delivery.
  - HTTP has no ability to try again or QoS features. If developers need guaranteed message delivery, they have to implement it themselves.

- **Extra libraries:**
  - MQTT works well on devices with limited memory, due, in part, to its small library requirement, which is only about 30 KB for the C client and 100 KB for the Java client.
  - HTTP does not require any libraries, but additional libraries of parsers for JavaScript Object Notation (JSON) or Extensible Markup Language (XML) are required if using the SOAP or Representational State Transfer (RESTful) style of web services.
Data distribution:

- MQTT includes a built-in distribution mechanism, supporting the 1 to 0, 1 to 1, and 1 to many distribution models.
- HTTP is point-to-point and has no built-in distribution feature. Developers must create their own distribution mechanism or adapt common techniques, such as long-polling or by using, for example, the Comet programming model.

An actual example occurred at Facebook with their Facebook Messenger\(^5\) product. Engineers implemented the MQTT protocol instead of HTTP, because the users experienced latency problems when sending messages. Message delivery was reliable but slow with HTTP. A new mechanism was needed for maintaining a persistent connection with the messaging servers without consuming too much battery power. This is critical to users of the company’s social networking site, because so many use battery-powered mobile devices. The company’s developers solved the problem by using the MQTT protocol. By maintaining an MQTT connection and routing messages through its chat pipeline, message delivery was accomplished at speeds of just a few hundred milliseconds, rather than multiple seconds.

1.3 MessageSight

IBM provides a suite of messaging products that helps businesses to meet the demands of today’s interconnected world. By using the IBM portfolio of messaging products, you can achieve a solution that extends from back-end enterprise applications to include millions of mobile users. MessageSight is one of the latest additions to the IBM messaging family. It is an appliance-based messaging server that is optimized to address the massive scale requirements of M2M and mobile use cases. MessageSight is designed to sit at the edge of the enterprise and can extend your existing messaging infrastructure or be used stand-alone.

MessageSight joins the IBM portfolio of middleware to help bridge back-end enterprise environments to remote smart clients as the planet becomes more digitally interconnected. This allows organizations to provide an interactive experience with users and offer real-time analytics of large data volumes (Figure 1-4).

![Figure 1-4 MessageSight extends your enterprise to remote wireless clients](image)

The MessageSight appliance is built for high performance to offer persistent, transactional messaging. This hardware is 2U form factor and can connect to a 1 GbE, 10 GbE, or 40 GbE network. MessageSight includes built-in security, and permits integration with external Lightweight Directory Access Protocol (LDAP) security systems.

---

\(^5\) Building Facebook Messenger; Lucy Zhang, August 12, 2011: https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920
With MessageSight, devices can sense and respond to data coming from the edge of your enterprise. Messaging connectivity can be performed using MQTT, Java Message Service (JMS), or IBM WebSphere MQ. The ability to connect through MQTT makes the appliance ideal for use with mobile clients. Administration can be done by using the web GUI or a command-line interface (CLI). The high availability pair takes the form of a primary-standby configuration. The primary node continually replicates both the message store and the appliance configuration information to the standby node. If the primary node fails, the standby node has the latest data that is needed for applications to continue messaging services. Standby node does not accept connections from application clients or provide messaging services until a failover operation occurs.

Mobile application support is provided by MessageSight, and so, it can handle massive scaling of concurrent device connectivity and communication, offering high performance messaging. The appliance is demilitarized zone (DMZ) ready, allowing you to securely extend your existing messaging enterprise. The device can act as the gateway for business data that flows in and out of your enterprise network.

MessageSight is developer friendly, designed for easy deployment and easy integration. This book explores how MessageSight can be integrated with other IBM offerings to deliver a complete messaging solution that encompasses remote mobile clients and back-end enterprise systems.

The MessageSight appliance can be used either with the MQTT protocol for low latency publishing and subscribing (ideal for machine-to-machine (M2M)), or with the Java Message Service (JMS) to transfer messages received from remote clients to back-end applications.

Figure 1-5 on page 14 shows examples of how clients who are connected to MessageSight can interface with WebSphere MQ and other back-end applications. Here, you can see that MessageSight connects many users and devices on the Internet to services that are deployed on an intranet. The users, devices, and services interact with each other by exchanging messages through the appliance.
MessageSight provides solutions to the following modern day challenges:

- A tremendous increase in the use of smartphones and tablets

  The increase in the number of smartphones and tablets is creating many endpoints. Consumers expect near real-time communication between their devices and applications. Building these applications relies on a scalable, bidirectional communication infrastructure. Emerging standards, such as HTML5 web sockets, provide the basis for building rich mobile, intranet, and Internet applications. MessageSight is a highly scalable middleware messaging product that provides the full-duplex web communication that is required for rich Internet, intranet, and mobile applications.

- Device-to-device communication

  The Internet is no longer just about web browsing. It is becoming a mesh of devices, such as sensors, monitors, machines, and cars, and these devices are becoming interconnected. Each device node attempts to publish data, consume data, or both in near real time. Applications attempt to consume data from these nodes, send data, or both. The IoT provides new challenges for traditional messaging infrastructures in terms of numbers of connected devices and the associated volume of messages.

  Systems already exist that understand what actions to take based on the status of remote devices. However, communicating that status to the system has been a challenge, particularly if the network is constrained or if the device lacks the computational power required for traditional messaging.
With MQTT, smart energy meters, industrial control systems, satellite receivers, healthcare monitoring devices, and sensors on everything from planes to trains to automobiles can communicate with each other and with other systems or applications.

1.4 Benefits of using MQTT

Using the MQTT protocol extends enterprise messaging to tiny sensors and other remote telemetry devices that might otherwise be unable to communicate with a central system or that might be reached only through the use of expensive, dedicated networks. Network limitations can include limited bandwidth, high latency, volume restrictions, fragile connections, or prohibitive costs. Device issues can include limited memory or processing capabilities or restrictions on the use of third-party communication software. In addition, certain devices are battery-powered, which puts additional restrictions on their use for telemetry messaging.

MQTT was designed to overcome these limitations and issues and includes the following underlying principles:

- **Simplicity:** The protocol was made open so that it can be integrated easily into other solutions.
- **Use of the publish/subscribe model:** The sender and the receiver are decoupled. Therefore, publishers do not need to know who or what is subscribing to messages, and vice versa.
- **Minimal maintenance:** Features, such as automated message storage and retransmission, minimize the need for administration tasks.
- **Limited on-the-wire footprint:** The protocol keeps data that is overhead to a minimum on every message.
- **Continuous session awareness:** By being aware of when sessions have terminated, the protocol can take action accordingly, thanks in part to a will feature.
- **Local message processing:** The protocol assumes that remote devices have limited processing capabilities.
- **Message persistence:** Through the designation of a specific QoS, the publisher can ensure delivery of the most important messages.
- **Agnostic regarding data type:** The protocol does not require that the content of messages is in any particular format.
Table 1-2 lists potential scenarios where MessageSight and the MQTT protocol might be used to improve communication to and from remote devices or applications.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Key industries</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated metering</td>
<td>Chemical and petroleum</td>
<td>A homeowner uses smart metering to obtain a more accurate view of how each household appliance consumes electricity and how to make changes.</td>
</tr>
<tr>
<td></td>
<td>Energy and utilities</td>
<td>An oil company monitors the gasoline sump levels at a gas station in real time and schedules a gasoline delivery when the sumps are nearing empty.</td>
</tr>
<tr>
<td>Distribution supply chain and logistics</td>
<td>Retailers</td>
<td>A shipping company gains customer loyalty by providing real-time, detailed tracking information for cargo.</td>
</tr>
<tr>
<td></td>
<td>Distributors</td>
<td>A trucking company cuts costs using remote fleet monitoring, which enables more efficient use of each truck's capacity on every run.</td>
</tr>
<tr>
<td></td>
<td>Consumer products</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transportation</td>
<td></td>
</tr>
<tr>
<td>Industrial tracking and visibility</td>
<td>Automotive</td>
<td>A manufacturing company automates inventory checking to improve the management of stock and to optimize production rates.</td>
</tr>
<tr>
<td></td>
<td>Industrial manufacturing</td>
<td>An automobile company uses RFID tracking to obtain real-time details about the current stage of assembly of each new vehicle as it moves through the assembly line.</td>
</tr>
<tr>
<td></td>
<td>Aerospace</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Defense</td>
<td></td>
</tr>
<tr>
<td>Health care quality and resource tracking</td>
<td>Pharmaceutical</td>
<td>A medical clinic remotely tracks the vital signs of at-risk patients to help prevent sudden crises that might arise after a patient goes home.</td>
</tr>
<tr>
<td></td>
<td>Medical research</td>
<td>A research team monitors chemical reactions in a remote laboratory and alerts the chemist when a particular result or stage of development is achieved.</td>
</tr>
<tr>
<td></td>
<td>Hospitals</td>
<td>A waste management company can be notified when the trash can is full and can be notified for pickup.</td>
</tr>
<tr>
<td></td>
<td>Nursing homes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Waste management</td>
<td></td>
</tr>
<tr>
<td>Location awareness and safety</td>
<td>Chemical and petroleum</td>
<td>A gas company improves pipeline monitoring by tripling the number of remote control devices on the route from 4,000 to 12,000 devices.</td>
</tr>
<tr>
<td></td>
<td>Energy and utilities</td>
<td>A government agency improves its early-warning system by placing remote sensors on dams and elsewhere in flood-prone regions.</td>
</tr>
<tr>
<td></td>
<td>Homeland defense</td>
<td></td>
</tr>
<tr>
<td>Executive alerts</td>
<td>Insurance</td>
<td>A bank retains more customers by monitoring all account-closing inquiries, analyzing this data, and contacting customers who might be considering closing their accounts.</td>
</tr>
<tr>
<td></td>
<td>Banking</td>
<td>An insurance company sends claims adjusters to collect damage reports at disaster sites and collects the data in its central servers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.5 Where to use MQTT

In today's world, there are many practical instances where MQTT and MessageSight implementations might offer a solution for an integrated system that allows users to interact using mobile devices. Here, we examine a few possible scenarios.

1.5.1 Connected car

MessageSight can enable customers to connect and interact with their cars. A connected car can use the MQTT protocol to send messages from the car to MessageSight and then notify the customer, as shown in Figure 1-6. For example, a connected car is one that is able to send diagnostic information to the automobile company. Also, the car can receive messages that might range from remotely locking the car to a request to send its current location.

MQTT and MessageSight facilitate message routing, allowing the car to send diagnostic information to the automobile company. The car basically acts as a rolling sensor platform that publishes telemetric events. The MQTT protocol is used to transport data from the car sensors to the automobile company. This data can be analyzed using predictive maintenance analytics, acting as a virtual mechanic, which then sends a notification to the customer that service is needed before a component fails.

Also, the car can receive messages ranging from remotely locking the car, setting the climate controls (heat or air), or requesting that the car send its current location. Most existing connected car solutions have previously taken a mobile response time measured in tens of seconds. Tests with MessageSight using MQTT have shown the response time to be less than one second, equal to the time it takes to push a key fob.

1.5.2 Connected city

If we take the connected car scenario a step further, we can see how having the MQTT messaging features available within many cars can effectively translate into a connected city. If a car is involved in an accident that caused the airbag to deploy, it can trigger an event to be published. Publish and subscribe messaging allows different users to receive the alert that an accident has happened. It can be generic to inform other drivers of the location of the accident, or it can be specific to route only to the car owner's family, and so on. The alert can be sent to emergency services to alert police or medics about the accident (Figure 1-7 on page 18).
In the connected city example depicted in Figure 1-7, it is easy to see how millions of cars sending messages can create a massive amount of data to be processed. To date, capturing and analyzing this amount of data in real time was a technical challenge. Using MessageSight in conjunction with IBM InfoSphere® Streams, stream computing helps to alleviate this dilemma by allowing real-time analytics to be performed on big data.

1.5.3 Connected home

The interactive user experience can also apply to a connected home. Figure 1-8 shows a scenario in which changing a channel on the TV creates a message that is sent back to the data center. In turn, this determines how advertising might be catered specifically to the consumer currently watching TV.

Other convenience features of the connected home are that a homeowner can adjust the thermostat or unlock a door using a mobile device. These types of features not only offer convenience; they also help to contribute to a Smarter Planet environment by being able to lower utility usage, as needed. The ability to remotely manage door locks and utilities can apply to a rental or vacation property, as well.
1.5.4 Connected consumers

*Connected consumers* can be provided with a unique shopping experience tailored to their location and shopping preferences. Bidirectional communication between a retailer’s back-end systems to the customer’s mobile devices allows retailers to provide notifications to consumers based on the customer’s proximity to the store (Figure 1-9).

![Connected consumers](image)

*Figure 1-9  Connected consumers*

For example, if a customer is browsing a product at home on a mobile device, then enters a retail store, the retailer’s mobile application can be used to find where the product is located in the store. The retailer might even want to push a notification for a sale or discount on that or a similar product when the customer is browsing in the store.

The retailer can enable business rules to handle database calls, analytics, pricing, and so on to cater to notifications that are based on the individual consumer, even the consumer’s current geographic location. The retailer’s central office is able to monitor the millions of messages using MessageSight.
Getting started with MQTT

This chapter introduces the MQTT protocol and later describes the key features in this protocol. It also provides a few examples of MQTT servers and MQTT clients, with usage examples of Eclipse Paho Java and JavaScript MQTT clients.

This chapter includes the following topics:

- 2.1, “MQTT concepts” on page 22
- 2.2, “MQTT server” on page 25
- 2.3, “MQTT clients” on page 26
2.1 MQTT concepts

This section describes the MQTT publish/subscribe messaging model and introduces the critical concepts involved in MQTT client programming.

2.1.1 MQTT messaging

The popularity of MQTT-based messaging stems from the simple way it allows information to be published or subscribed to, without the need to know who or what is sending or receiving the information. This simplicity allows each message to be quite small in size, therefore reducing demands on the network and on the remote monitoring devices from which many MQTT messages emanate.

Publish/subscribe pattern

MQTT uses a publish/subscribe messaging pattern that enables a loose coupling between the information provider, called the publisher, and consumers of the information, called subscribers. This is achieved by introducing an MQTT server between the publisher and the subscribers; Figure 2-1 illustrates the publish/subscribe example.

Compared with the traditional point-to-point pattern, the advantage of the publish/subscribe model is that the publishing device or application does not need to know anything about the subscribing one, and vice versa. The publisher simply sends the message with an identifier that denotes its topic, or subject area. The MQTT server then distributes the message to all applications or devices that have chosen to subscribe to that topic. In this way, the publish/subscribe pattern turns traditional point-to-point messaging into a multicast of content-based communications.

![Publish/subscribe example](image)
Figure 2-1 on page 22 shows a simple publish/subscribe configuration. Each publisher and subscriber only send and receive their specific information, and the MQTT server is positioned between them and routes each message in the correct direction based on its topic designation.

**MQTT client**

The MQTT client is a program or device that implements and uses the MQTT protocol. An MQTT client always establishes the network connection to the MQTT server. MQTT clients can act as a publisher or a subscriber. An MQTT client can perform these functions:

- Publish application messages that other MQTT clients might be interested in
- Subscribe to request application messages that it is interested in receiving
- Unsubscribe to remove a request for application messages
- Disconnect from the MQTT server

**MQTT server**

The MQTT server is a program or device that acts as an intermediary between MQTT clients that publish application messages and MQTT clients that have made subscriptions. An MQTT server can perform these functions:

- Accepts network connections from clients
- Accepts application messages published by clients
- Processes subscribe and unsubscribe requests from MQTT clients
- Forwards application messages that match MQTT client subscriptions

**Topics, trees, and strings**

Message distribution in MQTT-based systems depends on the designation of specific topics, topic trees, and topic strings.

**Topics**

Publishers are responsible for classifying their message subjects into *topics*. A topic defines the content of a message or the subject area under which the message can be categorized. In our scenario, the simplest example of a topic is the single word *sports*. Topics are important because, when messages in point-to-point systems are sent to specific destination addresses, messages in publish/subscribe systems, such as MQTT, are distributed based on each subscriber’s choice of topics. By subscribing to a particular topic, the subscriber is signing up to receive every message published to that topic from any publisher.

**Topic trees**

Typically, topics are organized hierarchically into *topic trees* that use the forward slash (/) character to create subtopics in the topic string. In our scenario, an example of a simple topic tree is `sports/tennis/wimbledon`.

**Topic strings**

A *topic string* is a character string that identifies the topic of a publish/subscribe message. Topic strings can contain either of two wildcard schemes to allow subscribers to match patterns within strings defined by message publishers:

- **Multilevel**: The wildcard character number sign (#) is used to match any number of levels within a topic. For example, subscribers to `sports/tennis/#` receive all messages that are designated for the topics `sports/tennis/wimbledon` and `sports/tennis/usopen`.¹

2.1.2 MQTT client programming

Several key concepts for MQTT client programming are provided in this section.

**Client identifier**
The client identifier is a 23-byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. To keep the identifier short and unique, developers typically introduce an identifier generation mechanism, such as creating it from the 48-bit device message authentication code (MAC) address. If transmission size is not a critical issue, the application might use the remaining 17 bytes to make the address easier to administer by inserting some human-readable text in the identifier, for example.

**Retained publications**
Publications (messages) for a given topic can be retained and delivered when new subscribers sign up for the topic. Publishers must set a retention attribute for each message; a setting of true retains the message and a setting of false establishes that the message will not be retained for future delivery. When a publication to be retained is created or updated, it is given a quality of service (QoS) designation of at least once (QoS=1) or exactly once (QoS=2). If a publication is sent with a QoS setting of at most once (QoS=0), a nonpersistent retained publication is automatically created and the publication is not retained if the queue manager stops.

Use retained publications to record the latest value of a measurement. New subscribers to the retained topic immediately receive the most recent value of the measurement. If no new measurements have been taken since the subscriber last subscribed to the publication topic, the subscriber still receives the most recent retained publication on the topic the next time the subscriber connects.

**Stateless and stateful sessions (cleanSession)**
The MQTT server identifies the client connection using the client identifier. The server checks whether session information has been saved from a previous connection to the server. The cleanSession parameter in the connection options indicates whether the connection is stateless or stateful. If a previous session still exists, and cleanSession=true, the previous session information at the client and server is cleared. If cleanSession=false, the previous session is resumed. If no previous session exists, a new session is started. The default value of cleanSession is true.

For publications, the clean session setting only affects publications sent with designations of QoS=1 and QoS=2. Using cleanSession=true might result in losing a publication, because it drops all publications that have not been received.

For subscriptions, if cleanSession=true, any old subscriptions for the client are removed when the client connects. Any new subscriptions the client makes during the session are removed when it disconnects.

**Last will and testament**
If an MQTT client connection ends unexpectedly, the client can configure a last will and testament publication to a topic. The client must predefine the content of the publication, and the topic to send the publication to. The last will and testament is a connection property that must be set before connecting to the client.
More details about the MQTT protocol are provided in Appendix A, “The MQTT protocol” on page 235.

### 2.2 MQTT server

An MQTT server implements the MQTT protocol and mediates communication between MQTT client applications, such as those running in remote sensors and other devices, and the enterprise integration layer.

#### 2.2.1 IBM MessageSight

IBM MessageSight delivers massive scale communication to extend the existing enterprise to include interactions from remote clients, such as mobile phones, MQTT-enabled sensors, machines, and other applications. The IBM MessageSight appliance delivers performance and scalability, enabling organizations to meet the demands of an always-connected world, and users who want an interactive experience. IBM MessageSight is the MQTT server used in all the scenarios in this book. More details about IBM MessageSight are available in Chapter 2, “Getting started with MQTT” on page 21.

You can download the developer's evaluation image of IBM MessageSight v1.0.0.1 from the following website:

http://ibm.co/1qzajLo

#### 2.2.2 IBM WebSphere MQ

IBM WebSphere MQ began including built-in support for MQTT, through the WebSphere MQ Telemetry component, starting with versions 7.0.1 and 7.1. The WebSphere MQ Telemetry component is implemented by the WebSphere MQ Extended Reach (MQXR) service. This MQXR service includes a Java-based broker, which enables delivery of MQTT messages by connecting MQTT clients to WebSphere MQ queue managers.

An evaluation version of WebSphere MQ is available at this website:


#### 2.2.3 Really Small Message Broker and MQ Telemetry daemon for devices

Really Small Message Broker (RSMB) is a small, no-charge C implementation of an MQTT broker. It also is an advanced MQTT V3 client application. Developers usually use it as a hub to store and forward messages from MQTT clients to a back-end MQTT broker. RSMB, which was first released in 2003, can be downloaded as a stand-alone client here:


#### 2.2.4 Mosquitto

Mosquitto is a small, no-cost, open source implementation of an MQTT server that supports the MQTT V3.1 protocol. Mosquitto replicates the functionality of Really Small Message Broker.
For more information about Mosquitto or to download the Mosquitto broker, see this website:
http://mosquitto.org/

### 2.3 MQTT clients

An MQTT client collects information from a telemetry device, connects to an MQTT server, and uses a topic string to publish the information in a way that allows other clients or applications to retrieve it. An MQTT client also can subscribe to topics, receive publications associated with those topics, and issue commands to control the telemetry device.

Client libraries can simplify the process of writing MQTT client applications. For example, the Eclipse Paho project provides Java and JavaScript client libraries that can be used to enable the MQTT V3 protocol for use on a number of platforms. When these libraries are incorporated into MQTT applications, a fully functional MQTT client can be created with just a few lines of code.

#### 2.3.1 Eclipse Paho clients

The Eclipse Paho project was developed to provide scalable open source implementations of open and standard messaging protocols for various emerging applications in the machine-to-machine (M2M) and Internet of Things (IoT) spaces. One of the major objectives of the Eclipse Paho project is to provide an effective level of decoupling between devices and applications.

The Eclipse Paho project initially started with MQTT publish/subscribe client implementations for use on embedded platforms, but in the future will provide corresponding server support as determined by the community. The current efforts of the Eclipse Paho project are to support the requirements of M2M integration with web and enterprise middleware and applications.

For M2M devices and client developers to integrate, develop, and test messaging components end to end, the Eclipse Paho project provides a framework to support testing and development of end-to-end device connectivity with a server. The MQTT client examples demonstrated in this chapter use the Java and JavaScript client libraries from the Eclipse Paho project.

The Eclipse Paho MQTT clients can be downloaded from this website:
http://www.eclipse.org/paho/download.php

#### 2.3.2 IBM Mobile Messaging and M2M Client Pack MA9B

IBM Mobile Messaging and machine-to-machine (M2M) Client Pack MA9B provide two Java client libraries. To read more details about MA9B, see the following website:
http://www-01.ibm.com/support/docview.wss?uid=swg27038199

**Java client: org.eclipse.paho.client.mqttv3**

The Java client library is in the ../SDK/clients/java directory of the MA9B client pack. This Java library implements the client portion of the MQTT V3.1 protocol implementation. The Java client runs on any suitable Java Runtime Environment (Java 1.5 and later), including Android. Separate interfaces are provided for synchronous and asynchronous styles of operation.
The synchronous application programming interface (API) supports a style where an operation only returns to the caller when the operation has completed. This is a traditional style, which might be used to implement a simple client. However, the blocking nature of this API limits its usage in environments where threads are not allowed to block, or when high performance is required or large numbers of clients are required.

The asynchronous API supports a different style in which a call returns immediately. The application can then either be notified through a callback when the operation completes, or can use a token returned to the application to block until the operation completes. This style of API is better suited to mobile, event-oriented, and high-performing applications. Applications where it might take time to complete an operation, or where a thread must not be blocked, are good candidates for the asynchronous API.

This client is a version of the open source MQTT client that is available from the Eclipse Paho project.²

Java client: com.ibm.micro.client.mqttv3

The Java client library is in the ../SDK/clients/java directory of the MA9B client pack. This Java library implements the client portion of the MQTT V3.1 protocol. An interface is provided for the synchronous style of operation. This is the same interface that was first included with WebSphere MQ V7.0.1, and it is suitable for use with existing applications. This interface has been stabilized.

For new applications, use the client in the org.eclipse.paho.client.mqttv3 package. To use this library, the Eclipse Paho version of the client (org.eclipse.paho.client.mqttv3v3.jar) must be included on the class path.

2.3.3 Preparation

The Java and JavaScript clients from the Eclipse Paho project are distributed as source code. Therefore, developers need to compile these clients on their own. The Eclipse Paho client source code is in a Git repository, and Git must be installed on the system before the compiling process is begun.

Preparing the Java client

The build script of the Eclipse Paho Java client requires the Apache Maven automated software build tool:

1. Download and install Apache Maven from the following website:
   http://maven.apache.org/download.cgi

2. Start at the console and enter the command shown in Example 2-1 to get the source code from the Git repository.

   Example 2-1  Clone the MQTT Java client on a local system

   git clone http://git.eclipse.org/gitroot/paho/org.eclipse.paho.mqtt.java.git

   We used a Microsoft Windows 7 machine to download the Java client.

² The Paho project: http://www.eclipse.org/paho/
3. Next, initiate the Maven command as shown in Example 2-2.

   **Example 2-2  Build the MQTT Java client**

   ```
   cd org.eclipse.paho.mqtt.java.git
   mvn package -DskipTests
   ```

4. This will build the client library without running the tests. The jars for the library, source, and Javadoc are in `org.eclipse.paho.client.mqttv3/target`. Figure 2-2 shows the target folder containing the Java library for Eclipse Paho.

![Figure 2-2  Eclipse Paho Java client built using Maven](image1.png)

5. When the build is complete, Maven shows the build output as shown in Example 2-3.

   **Example 2-3  Maven build log**

   ```
   [INFO] ------------------------------------------------------------------------
   [INFO] Building Eclipse Paho 1.0.0
   [INFO] ------------------------------------------------------------------------
   [INFO] Reactor Summary:
   [INFO] [INFO] org.eclipse.paho.client.mqttv3 ..................... SUCCESS [01:29 min]
   [INFO] org.eclipse.paho.client.mqttv3.test .................. SUCCESS [ 32.801 s]
   [INFO] org.eclipse.paho.client.eclipse.view ................. SUCCESS [ 0.788 s]
   [INFO] org.eclipse.paho.client.eclipse.feature .......... SUCCESS [ 0.087 s]
   [INFO] Paho P2 Repository .................................. SUCCESS [ 1.893 s]
   [INFO] org.eclipse.paho.mqtt.util ......................... SUCCESS [ 37.279 s]
   [INFO] org.eclipse.paho.ui.core ......................... SUCCESS [ 3.305 s]
   [INFO] org.eclipse.paho.ui.app .......................... SUCCESS [ 39.849 s]
   [INFO] org.eclipse.paho.ui ................................ SUCCESS [ 0.001 s]
   [INFO] Eclipse Paho ................................... SUCCESS [ 0.000 s]
   [INFO] ------------------------------------------------------------------------
   [INFO] BUILD SUCCESS
   [INFO] ------------------------------------------------------------------------
   [INFO] Total time: 07:03 min
   [INFO] Finished at: 2014-07-23T06:11:49-05:00
   [INFO] Final Memory: 105M/209M
   [INFO] ------------------------------------------------------------------------
   ```
Preparing the JavaScript client
The JavaScript client from the Eclipse Paho project is a browser-based library that uses websockets to connect to an MQTT v3.1 server. A compressed file containing the full version and a minimized version of the JavaScript client can be downloaded from the following website:

http://download.eclipse.org/paho/1.0/paho.javascript-1.0.0.zip

Alternatively, the JavaScript client can be downloaded directly from the project's Git repository:


The build script of the Eclipse Paho JavaScript client requires the Apache Maven automated software build tool:

1. Download and install Apache Maven from the following website:

http://maven.apache.org/download.cgi

2. Start at the console and enter the command shown in Example 2-4 to get the source code from the Git repository.

Example 2-4  Clone the MQTT JavaScript client on a local system

```bash
git clone http://git.eclipse.org/gitroot/paho/org.eclipse.paho.mqtt.javascript.git
```

We used a Microsoft Windows 7 machine to download the Java client.

3. Next, initiate the Maven command as shown in Example 2-5.

Example 2-5  Maven command

```bash
cd org.eclipse.paho.mqtt.javascript.git
mvn
```

4. The output is copied to the target directory.

2.3.4 Building the sample MQTT application

Figure 2-1 on page 22 showed an example of MQTT client and server application publishing and subscribing on tennis events such as Wimbledon and US Open. The publisher publishes on the topics `sports/tennis/wimbledon` and `sports/tennis/usopen`, and the subscriber can subscribe on the topic `sports/tennis/+` to subscribe to all the tennis events or subscribe on individual events using topics, such as `sports/tennis/wimbledon` and `sports/tennis/usopen`.

This use case is a classic publish/subscribe application, as illustrated in Figure 2-3 on page 30.
The next two sections describe how to build an MQTT client application that can publish and subscribe to topics. Examples are provided for Java, JavaScript, and HTML5.

The following general steps are performed when creating the applications:
1. Create an instance of an MQTT client.
2. Prepare connection options and connect to the MQTT server using these options.
3. Publish messages to topics, and subscribe to topics.
4. Disconnect from the MQTT server.

### 2.3.5 MQTT publisher and subscriber in Java

This section describes how to use Java to build an MQTT client application to publish and subscribe to topics. An Eclipse runtime is used to build this client.

**Publisher application**
The publisher application consists of two Java classes, as described in Table 2-1.

<table>
<thead>
<tr>
<th>Class name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQTTPublisherConstants.java</td>
<td>This class defines the constants for publisher applications, such as the address of the MQTT server, the client ID of the publisher, the QoS, and other connection parameters. Example 2-6 shows the implementation of this class.</td>
</tr>
<tr>
<td>MQTTPublisher.java</td>
<td>This class is used to connect to an MQTT server, publish a message to a topic, and disconnect from the server. Example 2-7 on page 31 shows the implementation of this class.</td>
</tr>
</tbody>
</table>

Example 2-6 on page 31 shows the Java class MQTTPublisherConstants. It is used to set the connection and other publication-related configuration information.
Example 2-6  MQTTPublisherConstants.java

```java
package com.ibm.redbook.sports;
public final class MQTTPublisherConstants {

    public static final String TCPADDRESS = "tcp://messagesight.usar.ibm.com:1883";
    public static final String CLIENTID = "sportnewspublisher";
    public static final int SLEEPTIMEOUT = 10000;
    public static final int QOS0 = 0;
    public static final boolean RETAINED = false;
    public static final String TOPICWIMBLEDON = "sports/tennis/wimbledon";
    public static final String TOPICUSOPEN = "sports/tennis/usopen";
    public static final String WIMBLEDON_PAYLOAD = "News for Wimbledon 2014";
    public static final String USOPEN_PAYLOAD = "News for USOPEN 2014";
}
```

Example 2-7 shows the Java class MQTTPublisher. This class uses the Eclipse Paho Java APIs for the publication of messages about different topics.

Example 2-7  MQTTPublisher.java

```java
package com.ibm.redbook.sports;

import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttDeliveryToken;

public class MQTTPublisher {

    public static void main(String[] args) {
        try {
            //a. Create an instance of MQTT client
            MqttClient client = new MqttClient(MQTTPublisherConstants.TCPADDRESS,
                MQTTPublisherConstants.CLIENTID);

            //b. Prepare connection options
            //Use default connection options in this sample.
            //c. Connect to server with the connection options
            client.connect();

            //d. Publish message to topics
            MqttTopic topic = client.getTopic(MQTTPublisherConstants.TOPICWIMBLEDON);
            MqttMessage message = new MqttMessage(MQTTPublisherConstants.WIMBLEDON_PAYLOAD.getBytes());
            message.setQos(MQTTPublisherConstants.QOS0);
            System.out.println("Waiting for up to " + MQTTPublisherConstants.SLEEPTIMEOUT /
                1000 + " seconds for publication of " + message.toString() + " with QoS = " +
                message.getQos());

            System.out.println("On topic " + topic.getName() + " for client instance: " +
                client.getClientId() + " on address " + client.getServerURI() + ");
        }
    }
```
MqttDeliveryToken token = topic.publish(message);
token.waitForCompletion(MQTPublisherConstants.SLEEPTIMEOUT);

System.out.println("Delivery token " + token.hashCode() + " has been received: "+ token.isComplete());
//e. Disconnect to server
client.disconnect();

catch (Exception e) {
  e.printStackTrace();
}
}

---

**Details of the publisher application**

Parts of the MQTPublisher.java code require additional explanation:

1. Create a try-catch block, as shown here, to handle any checked exceptions (MqttException or its subclasses MqttPersistenceException and MqttSecurityException) that are thrown by the MQTT client:

   ```java
   try {
   } catch (Exception e) {
       e.printStackTrace();
   }
   ```

2. Create a new MqttClient instance using the following command:

   ```java
   MqttClient client = new MqttClient(MQTTPublisherConstants.TCPADDRESS,
   MQTPublisherConstants.CLIENTID)
   ```

   The default port of IBM MessageSight DemoMqttEndpoint is 1883. In this example, the default address, MQTPublisherConstants.TCPADDRESS, is set to tcp://messagesight.usar.ibm.com:1883.

   The client identifier, MQTPublisherConstants.CLIENTID, must be unique across all clients connecting to a server. For more information, see the explanation of the MQTT client identifier in “Client identifier” on page 24.

3. Optionally, the client can provide an implementation of the MqttClientPersistence interface to replace the default implementation. The default implementation stores messages that are awaiting delivery (such as messages with QoS designations of 1 or 2) as files in the current directory. If the client wants to change the directory where the files are stored, it can create an MqttDefaultFilePersistence instance and provide it as a third parameter for the constructor MqttClient.

   The MqttDefaultFilePersistence class needs to be configured as shown here and used when creating the MQTT client instance optionally. It stores messages in the local directory that is specified in the constructor:

   ```java
   MqttClientPersistence clientPersistence = new
   MqttDefaultFilePersistence("C:/MQTTPublisher/clientdir");
   client = new MqttClient("tcp://messagesight.usar.ibm.com:1883",
   "sportnewspublisher", clientPersistence);
   ```

   A subdirectory prefixed with the client ID is created in the specified directory and is used when sending and receiving QoS 1 and QoS 2 messages.
4. Connect the client to the server. If done as shown here, the default settings are used. A small message will be sent every 15 seconds to prevent the TCP/IP connection from being closed. The `cleanSession` variable is equal to `true`. The session is started without checking for the completion of previous publications. And, no last will and testament message is created for the connection.

```java
client.connect();
```

5. Create a topic to which to publish a message. In the code shown here, `MQTTPublisherConstants.TOPICWIMBLEDON` can be replaced with an actual topic string, such as `sports/tennis/wimbledon`:

```java
MqttTopic topic = client.getTopic(MQTTPublisherConstants.TOPICWIMBLEDON);
```

6. Create a message for publication, as shown here. The message in this sample, `MQTTPublisherConstants.WIBLEDON_PAYLOAD`, will display the latest news from Wimbledon. For simplicity, the message published here is fixed:

```java
MqttMessage message = new MqttMessage(MQTTPublisherConstants.WIMBLEDON_PAYLOAD.getBytes());
message.setQos(MQTTPublisherConstants.QOS0);
```

Because an MQTT message requires a byte array as input, `MQTTPublisherConstants.WIBLEDON_PAYLOAD` is converted to a byte array using the `getBytes` method, and the encoding is set to UTF-8. The QoS designation, which is set to 0 here, determines how reliably the message is transferred to the MQTT client and then between the MQTT client and the server.

7. Publish the message to the server, as shown here:

```java
MqttDeliveryToken token = topic.publish(message);
```

When the publish method returns, the message has been safely transferred to the MQTT client, but not yet transferred to the server. If the message has a QoS designation of 1 or 2, the message is stored locally, in case the client fails before delivery is completed. A delivery token is returned and used to check whether an acknowledgment has been received from the server yet. MQTT delivery tokens are unique to the MQTT client and enable users to monitor delivery of a published message.

8. Wait for an acknowledgment from the server confirming that the message has been delivered. The example provided here shows a timeout, without which the client might wait indefinitely. If using an asynchronous API for publishing, the program does not need to be blocked until it gets the acknowledgment from the server. The callback function will be called after successful message delivery.

```java
token.waitForCompletion(MQTTPublisherConstants.SLEEPTIMEOUT);
```

9. Disconnect the client from the server, as shown here. The client disconnects from the server and waits for any `MqttCallback` methods that are running to finish. The client then waits for up to 30 seconds to finish any remaining work. The client can specify a timeout as an additional parameter.

```java
client.disconnect();
```

**Compilation and execution**

This program can also be executed through Eclipse. Eclipse compiles the program automatically. Then, you can right-click the `MQTTPublisher.java` file, and then select Run As → Java Application.

After starting the `MQTTPublisher` application, a message displays in the console window, similar to the messages shown in Example 2-8 on page 34, which means that the message was delivered to the MQTT server successfully.
Example 2-8  Message delivered successfully to the MQTT server
Waiting for up to 10 seconds for publication of "Message for USOPEN 2014" with QoS = 0
On topic "sports/tennis/usopen" for client instance: "sportnewspublisher" on address tcp://messagesight.usar.ibm.com:1883"
Delivery token "-2051063733" has been received: true

Subscriber application
The subscriber application consists of three Java classes, as described in Table 2-2.

Table 2-2  Subscriber application Java classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQTTSubscriberConstants.java</td>
<td>This class defines the constants for publisher applications, such as the address of the MQTT server, the client ID of the publisher, the QoS, and other connection parameters. Example 2-9 shows the implementation of this class.</td>
</tr>
<tr>
<td>MQTTSubscriberCallback.java</td>
<td>This class implements the MqttCallback interface and is set to work with an MQTT client. Message arrival acknowledgments are received in a callback function that is defined in the MqttCallback interface. Example 2-10 shows the implementation of this class.</td>
</tr>
<tr>
<td>MQTTSubscriber.java</td>
<td>This class creates the subscription and waits for matching publications. The handling of the actual message is delegated to the Callback class, which is defined in MQTTSubscriberCallback.java. In Example 2-10 on page 35, MQTTSubscriber.java uses an asynchronous programming model, which means it does not need to wait for a message to arrive and block the execution of the program. Example 2-11 on page 36 shows the implementation of this class.</td>
</tr>
</tbody>
</table>

Example 2-9 shows the Java class MQTTSubscriberConstants and how it is used to set the connection and other subscription-related configuration information.

Example 2-9  MQTTSubscriberConstants.java

package com.ibm.redbook.sports;

public final class MQTTSubscriberConstants {

    public static final String TCPADDRESS = "tcp://messagesight.usar.ibm.com:1883";
    public static final String CLIENTID = "sportnewssubscriber";
    public static final boolean CLEANSESSION = true;
    public static final int KEEPALIVEINTERVAL = 20;
    public static final int QOS0 = 0;
    public static final String TOPICWIMBLEDON = "sports/tennis/wimbledon";
    public static final String TOPICUSOPEN = "sports/tennis/usopen";
    public static final String TOPICTENNIS = "sports/tennis/";

}
Example 2-10 shows the Java class MQTTSsubscriberCallback. This class implements the MqttCallback interface and is set to work with an MQTT client. Message arrival acknowledgments are received in a callback function that is defined in the MqttCallback interface.

Example 2-10  MQTTSsubscriberCallback.java

```java
package com.ibm.redbook.sports;

import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttException;

public class MQTTSsubscriberCallback implements MqttCallback {

    private String instanceData = "";
    public MQTTSsubscriberCallback(String instance) {
        instanceData = instance;
    }

    public void connectionLost(Throwable cause) {
        System.out.println("Connection lost on instance "+ instanceData + "," + cause.getMessage() + ", Reason code "+
        ((MqttException)cause).getReasonCode() + ", Cause "+
        ((MqttException)cause).getCause() + "");
        cause.printStackTrace();
    }

    public void deliveryComplete(IMqttDeliveryToken token) {
        try {
            System.out.println("Delivery token " + token.hashCode() + ", received by instance "+ instanceData + ",");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void messageArrived(String topic, MqttMessage message) throws Exception {
        try {
            System.out.println("Message arrived: " + message.toString() + ", on topic ", topic.toString() + ", for instance "+ instanceData + ",");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Example 2-11 on page 36 shows the class MQTTSsubscriber. This class creates the subscription and waits for matching publications. The handling of the actual message is delegated to the Callback class, which is defined in MQTTSsubscriberCallback.java in Example 2-10. MQTTSsubscriber.java uses an asynchronous programming model, which means it does not need to wait for a message to arrive and block the execution of the program.
Example 2-11  MQTTSubscriber.java

```java
package com.ibm.redbook.sports;

import java.util.Scanner;

import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;

public class MQTTSubscriber {
    public static void main(String[] args) {
        try {
            //a. Create an instance of MQTT client
            MqttClient client = new MqttClient(MQTTSubscriberConstants.TCPADDRESS,
                    MQTTSubscriberConstants.CLIENTID);

            //b. Prepare connection options
            MQTTSubscriberCallback callback = new
                    MQTTSubscriberCallback(MQTTSubscriberConstants.CLIENTID);
            client.setCallback(callback);
            MqttConnectOptions conOptions = new MqttConnectOptions();
            conOptions.setCleanSession(MQTTSubscriberConstants.CLEANSESSION);
            conOptions.setKeepAliveInterval(MQTTSubscriberConstants.KEEPALIVEINTERVAL);

            //c. Connect to broker with the connection options
            client.connect(conOptions);
            System.out.println("Subscribing to topic "+
                    MQTTSubscriberConstants.TOPICTENNIS + "+ for client instance "+
                    client.getClientId() + "+ using QoS " +
                    MQTTSubscriberConstants.QOS0 + ". Clean session is " +
                    MQTTSubscriberConstants.CLEANSESSION);

            //d. Subscribe interested topics.
            client.subscribe(MQTTSubscriberConstants.TOPICTENNIS,
                    MQTTSubscriberConstants.QOS0);

            System.out.println("Subscribing to topic "+
                    MQTTSubscriberConstants.TOPICTENNIS + "+ for client "+
                    client.getClientId() + " using QoS " +
                    MQTTSubscriberConstants.QOS0 + ". Clean session is " +
                    MQTTSubscriberConstants.CLEANSESSION);

            //e. Disconnect to broker
            client.disconnect();
            System.out.println("Subscriber ending");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
```
Details of the subscriber application

Parts of the MQTTSubscriber.java code require additional explanation:

- Create an instance of the Callback class as shown here and set it to the MQTT client that was declared in the publishing application. The Callback class implements the MqttCallback interface. One callback instance per client identifier is required. In this example, the constructor passes the client identifier to save as instance data:

```java
MQTTSubscriberCallback callback = new MQTTSubscriberCallback(MQTTSubscriberConstants.CLIENTID);
client.setCallback(callback);
```

The Callback interface includes the following methods:

- `public void messageArrived(String topic, MqttMessage message)`: Receives a publication that has been subscribed to.
- `public void connectionLost(Throwable cause)`: Called when the connection is lost.
- `public void deliveryComplete(IMqttDeliveryToken token)`: Called when a delivery token is received for a published message carrying a QoS designation of 1 or 2.

The Callback class in MQTTSubscriberCallback.java implements these three methods. When a message is delivered to the topic to which the program subscribes, it writes the message to the standard output device (typically the console).

- Create an MqttConnectOptions object to configure the connection properties of the MQTT client. In this example, the cleanSession and keepAliveInterval attributes are set:

```java
MqttConnectOptions conOptions = new MqttConnectOptions();
conOptions.setCleanSession(MQTTSubscriberConstants.CLEANSESSION);
conOptions.setKeepAliveInterval(MQTTSubscriberConstants.KEEPALIVEINTERVAL);
```

If the client uses the default MqttConnectOptions, or sets MqttConnectOptions.cleanSession to true before connecting the client, any old subscriptions for the client are removed when the client connects. Any new subscriptions the client makes during the session are removed when it disconnects.

If the client sets MqttConnectOptions.cleanSession to false before connecting, any subscriptions the client creates are added to all the subscriptions that existed for the client before it connected. All the subscriptions remain active when the client disconnects.

Set the cleanSession mode before connecting. The mode lasts for the whole session. To change this setting, disconnect and reconnect the client. If the client changes the mode from using cleanSession=false to cleanSession=true, all previous subscriptions for the client and any publications that have not been received are discarded.

Additional connection options are available, including MQTT connect timeout. With this option, when the client connects to the server it can set a connection timeout. This is important because in certain networks and operating systems, when a connection is attempted on a socket that is not reachable, the ensuing errors are not returned immediately. Using the method `setConnectionTimeout()` allows the developer to specify how long the client can wait before it decides that the server is not reachable.

- Create a subscription and set its QoS level. The example shown here uses an MqttClient.subscribe method that passes one topic string with a QoS option:

```java
client.subscribe(MQTTSubscriberConstants.TOPICTENNIS, MQTTSubscriberConstants.QOS0);
```

Each time MQTTSubscriber.java is executed, it creates a subscription. So, unless MQTTSubscriberConstants.TOPICTENNIS is changed each time, the same subscription will be re-created over and over again. In this example, the subscription topic is sports/tennis/# by default.
Wait for publications to arrive, or for quitting type q or Q on the command line to disconnect the client and exit the application, as shown here:

Scanner scanner = new Scanner(System.in);
for(String input=""; !input.equalsIgnoreCase("q");
    input = scanner.nextLine());

Compilation and execution
This program can also be executed through Eclipse. Eclipse compiles the program automatically. Then, you can right-click the MQTTSubscriber.java file, and then select Run As → Java Application.

After starting the MQTTSubscriber application, a message is seen in the console window, similar to the messages shown in Example 2-12, which means that the application started successfully and received a message sent by the publisher.

Example 2-12  Message delivered successfully to MQTT subscriber

<table>
<thead>
<tr>
<th>Message arrived: &quot;Message for USOPEN 2014&quot; on topic &quot;sports/tennis/usopen&quot; for instance &quot;sportnewssubscriber&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
<tr>
<td>Finished</td>
</tr>
</tbody>
</table>

The MQTT publisher and subscriber application for Java can be downloaded from the hyperlink included in Appendix B, “Additional material” on page 245.

2.3.6 MQTT publisher and subscriber in JavaScript

This section describes how to use JavaScript to build an MQTT client application to publish and subscribe to topics. A web browser can be used to execute and test this application.

The JavaScript MQTT publisher and subscriber application described in this section uses the Eclipse Paho JavaScript library. The JavaScript library consists of three objects, as described in Table 2-3 on page 39.
Table 2-3  Eclipse Paho project JavaScript client objects

<table>
<thead>
<tr>
<th>Name</th>
<th>Object type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messaging</td>
<td>Namespace</td>
<td>You can send and receive messages using web browsers. This programming interface lets a JavaScript client application use the MQTT V3.1 protocol to connect to an MQTT-supporting messaging server. The following functions are supported:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Connecting to and disconnecting from a server. The server is identified by its host name and port number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Specifying options that relate to the communications link with the server, for example, the frequency of keep-alive heartbeats, and whether Secure Sockets Layer (SSL)/Transport Layer Security (TLS) is required.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Subscribing to and receiving messages from MQTT Topics.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Publishing messages to MQTT Topics.</td>
</tr>
<tr>
<td>Messaging.Client</td>
<td>Class</td>
<td>This class contains methods that provide the functionality of the API, including the provision of callbacks that notify the application when a message arrives from or is delivered to the messaging server, or when the status of its connection to the messaging server changes.</td>
</tr>
<tr>
<td>Messaging.Message</td>
<td>Class</td>
<td>This class encapsulates the payload of the message, with various attributes associated with its delivery, in particular the destination to which it has been (or is about to be) sent.</td>
</tr>
</tbody>
</table>

Publisher application

The publisher application is implemented in the WebSocket/publisher.js file. Example 2-13 shows the implementation of this JavaScript application.

Example 2-13  ITSO tennis news JavaScript publisher

```javascript
var clientId = "sportnewspublisher";
var client;

function doConnect(form) {
    client = new Messaging.Client(form.serverName.value.trim(),
        parseFloat(form.serverPort.value), clientId);
    client.onConnectionLost = onConnectionLost;
    client.connect({onSuccess : onConnect});
}

function doPublish(form) {
    var message = new Messaging.Message(form.newsMessage.value);
    message.destinationName = "sports/tennis/wimbledon";
    form.newsMessage.value="";
    client.send(message);
}

function doDisconnect() {
    clearAll();
    client.disconnect();
}

function clearAll() {
    var form = document.getElementById("publisher");
    form.serverName.value = "";
    form.serverPort.value = "";
    form.newsMessage.value = "";
}
```
Details about the publisher application

To send and receive messages, we need to develop a Messaging client instance and connect it to a server. The client instance is constructed in the following manner:

```javascript
client = new Messaging.Client(form.serverName.value.trim(),
parseFloat(form.serverPort.value), clientId);
```

The client is created with the Domain Name System (DNS) host name and port to which it is going to connect. It also needs to know the unique client identifier that will identify the client to the server. In Example 2-13 on page 39, we used the host name and port number that are entered through the HTML web page so the client makes a connection back to the same server.

Next, the callbacks are set for a successful connection and when the connection is lost, this is coded using the following snippets:

- `client.connect({onSuccess : onConnect});`
- `client.onConnectionLost = onConnectionLost;`

These callbacks are driven when the connection is successfully established or when the connection is lost. Web browsers do not allow method calls to block so we need to rely on callbacks to notify us when an action is complete.

After the publisher application is connected to the server, a message can be created and published to a specific topic using the following code snippet:

```javascript
var message = new Messaging.Message(form.newsMessage.value);
message.destinationName = "sports/tennis/wimbledon";
client.send(message);
```

In this scenario, a message can be published on the topic `sports/tennis/wimbledon`. 

```javascript
form.serverName.disabled = false;
form.serverPort.disabled = false;
form.newsMessage.disabled = true;
form.publishButton.disabled = true;
form.disconnectButton.disabled = true;
}

// Messaging callbacks

function onConnect() {
    var form = document.getElementById("publisher");
    form.connected.checked = true;
    form.serverName.disabled = true;
    form.serverPort.disabled = true;
    form.newsMessage.disabled = false;
    form.publishButton.disabled = false;
    form.disconnectButton.disabled = false;
}

function onConnectionLost(responseObject) {
    var form = document.getElementById("publisher");
    form.connected.checked = false;
    if (responseObject.errorCode !== 0)
        alert(client.clientId + "\n" + responseObject.errorCode);
}
After the messages are published, the publisher client might disconnect from the server using the following code snippet:

```java
client.disconnect();
```

**Execute the publisher application**

An HTML-based application is developed that calls the JavaScript functions that are implemented in `publisher.js`. The HTML and JavaScript implementation for the publisher can be downloaded from the hyperlink included in Appendix B, “Additional material” on page 245.

Figure 2-4 shows the publisher application opened in a web browser for publishing tennis-related news.

![Figure 2-4 ITSO tennis news publisher application](image.png)

To execute this application, enter the MessageSight server name and the port number for endpoints, and click **Connect**. After the application is connected, the news publisher can publish related news messages, as shown in Figure 2-5 on page 42. To disconnect the application, click **Disconnect**.
Subscriber application

The subscriber application is implemented in the WebSocket/subscriber.js file. Example 2-14 shows the implementation of this JavaScript application.

Example 2-14  ITSO tennis news JavaScript subscriber

```javascript
var clientId = "sportnewssubscriber";
var client;

function doConnect(form) {
    client = new Messaging.Client(form.serverName.value.trim(),
        parseFloat(form.serverPort.value), clientId);
    client.onMessageArrived = onMessageArrived;
    client.onConnectionLost = onConnectionLost;
    client.connect({onSuccess : onConnect});
}

function doSubscribe(form) {
    client.subscribe("sports/tennis/+"),
    form.newsMessage.disabled = false;
}

function doDisconnect() {
    clearAll();
    client.unsubscribe("sports/tennis/+"),
    client.disconnect();
}
```
function clearAll() {
    var form = document.getElementById("subscriber");
    form.serverName.value = "";
    form.serverPort.value = "";
    form.newsMessage.value = "";
    form.serverName.disabled = false;
    form.serverPort.disabled = false;
    form.subscribeButton.disabled = true;
    form.newsMessage.disabled = true;
    form.disconnectButton.disabled = true;
}

function addText(form, message) {
    var newtext = message.payloadString;
    var existingText = form.newsMessage.value;
    form.newsMessage.value = existingText+newtext+"\n";
}

// Messaging callbacks

function onConnect() {
    var form = document.getElementById("subscriber");
    form.serverName.disabled = true;
    form.serverPort.disabled = true;
    form.connected.checked = true;
    form.subscribeButton.disabled = false;
    form.disconnectButton.disabled = false;
}

function onConnectionLost(responseObject) {
    var form = document.getElementById("subscriber");
    form.connected.checked = false;
    if (responseObject.errorCode !== 0)
        alert(client.clientId + "\n" + responseObject.errorCode);
}

function onMessageArrived(message) {
    var form = document.getElementById("subscriber");
    addText(form,message);
}

Details about the subscriber application
To send and receive messages, we need to develop a Messaging client instance and connect it to a server. The client instance is constructed in the following manner:

client = new Messaging.Client(form.serverName.value.trim(),
parseFloat(form.serverPort.value), clientId);

The client is created with the DNS host name and port to which it is going to connect. It also needs to know the unique client identifier that will identify the client to the server. In Example 2-14 on page 42, we used the host name and port number that are entered through the HTML web page so the client makes a connection back to the same server.
Next, the callbacks are set for a successful connection. When the connection is lost or when a message arrived for the application, this is coded using the following snippets:

```
client.connect({onSuccess : onConnect});
client.onConnectionLost = onConnectionLost;
client.onMessageArrived = onMessageArrived;
```

These callbacks are driven when the connection is successfully established, when the connection is lost, or when the message arrives for the application. Web browsers do not allow method calls to block so we need to rely on callbacks to notify us when an action is complete.

The subscriber application subscribes to a topic for receiving messages that are published on that topic. The following code snippet shows the subscription request for the topic filter "sports/tennis/+".

```
client.subscribe("sports/tennis/+”);
```

When the application is subscribed to a topic, any messages published on that topic will be sent to this application.

The application might choose to unsubscribe and disconnect from the server, and this can be done using the following code snippet:

```
client.unsubscribe("sports/tennis/+”);
client.disconnect();
```

**Execute the subscriber application**

An HTML-based application is developed that calls the JavaScript functions implemented in subscriber.js. HTML and JavaScript implementation for the subscriber can be downloaded from the hyperlink that is included in Appendix B, “Additional material” on page 245.

Figure 2-6 on page 45 shows the subscriber application opened in a web browser to subscribe for tennis-related news.
To execute this application, enter the MessageSight server name and port number for endpoints and click **Connect**. After the subscriber application is connected to MessageSight appliance, click **Subscribe** to subscribe on the tennis news topics. Any news publication published on the subscribed topic will be received by this application. Figure 2-7 on page 46 shows the news message received by the server. To disconnect the application, click **Disconnect**.
ITSO Tennis News Subscriber

This page is a complete example of a Web Messaging application written in JavaScript for subscribing Tennis related News

MQTT clients can be implemented in different programming languages. To read more about the MQTT specification, see Appendix A, “The MQTT protocol” on page 235.

Figure 2-7  ITSO tennis news subscriber application receiving wimbledon news
Overview of IBM MessageSight

Interactions with the Internet no longer happen solely by individuals who are using a computer. Users connect to the Internet with a variety of devices, such as mobile phones, sensors, and machines, because the ubiquity of TCP/IP over 3G and 4G cellular networks enables these various types of devices to send and receive data.

The Internet is transitioning to the Internet of Things (IoT), where things or machines, connect to machines (M2M), and interact with reduced, if any, human intervention. With the numbers of devices in use increasing, organizations require a scalable, reliable, and cost-effective solution for connecting these devices to their system of records.

The IBM MessageSight messaging appliance helps to deliver the performance, value, and simplicity that organizations need for accommodating this multitude of devices and processing large volumes of events in real time.

MessageSight extends existing messaging networks by adding the following characteristics:
- Fast transaction rates
- Consistent lower latency
- Extensive scaling in the number of concurrent devices that can be connected
- Suitable for deployment in a demilitarized zone (DMZ)

This chapter provides details about the following topics:
- 3.1, “Features of MessageSight” on page 48
- 3.2, “Messaging patterns of MessageSight” on page 50
- 3.3, “Install the MessageSight virtual appliance (for developers)” on page 54
- 3.4, “Overview of the MessageSight web UI” on page 57
- 3.5, “Overview of the MessageSight CLI” on page 70
- 3.6, “Message hubs, endpoints, and policies” on page 75
3.1 Features of MessageSight

Application owners and consumers require large-scale connectivity and a high-speed communication solution to enable the real-time capture of interactions between the multitude of connected devices and applications.

The explosion in the number of mobile devices, such as smartphones and tablets, is creating many endpoints. Consumers expect real-time communication between their devices and applications. Building these applications relies on a scalable, bidirectional communication infrastructure. Emerging standards, such as HTML5 web sockets, provide the basis for building rich mobile, intranet, and Internet applications.

By enabling the use of messaging protocols, such as MQTT, MessageSight is a highly scalable middleware messaging product that provides the full-duplex web communication that is required for these mobile, intranet, and Internet applications.

Note: MQTT is a messaging protocol designed for wireless networks and devices with which conserving device battery life, reducing network traffic, and delivering reliable messages over unreliable networks are key. MQTT is an open protocol with no-cost MQTT clients available for a wide range of mobile platforms.

For details about the MQTT protocol, see Appendix A, “The MQTT protocol” on page 235.

Figure 3-1 shows the wireless MQTT protocol as it connects to MessageSight, inside an enterprise network.

MessageSight offers the following features:

- **Providing a quick and simplified deployment**
  The appliance can be configured for a typical environment within about 30 minutes. The web user interface (UI) helps to guide administrators through the initial steps. Configuration is simple, using either the web UI or command-line interface (CLI). Administration is optimized for scale using a policy-based approach.

- **Enabling the messaging infrastructure for use in a DMZ**
  The MessageSight appliance is designed to sit at the edge of the enterprise, from where it can extend an existing messaging infrastructure or be used as a stand-alone appliance. MessageSight has no user accessible operating system and only accepts signed and encrypted firmware. Only configured services can enable listening ports with no routing
throughout the network interfaces. MessageSight consists of encrypted compact flash and storage media. The compact flash is locked down and tied to system use.

- **Providing security-rich messaging using a policy-based approach**
  Messaging policies allow you to filter for specific access. Options are available to add Secure Sockets Layer/Transport Layer Security (SSL/TLS), including Federal Information Processing Standard (FIPS) 140-2.

- **Employing open standards and protocols for greater flexibility**
  MessageSight supports the MQTT V3.1 specification, MQTT over HTML5 websockets, and Java Message Service (JMS 1.1) for inbound and outbound messaging. MessageSight also supports protocol mediation.

- **Providing high reliability and performance**
  High availability configurations are supported when using a pair of MessageSight appliances. In addition, high-message throughput for millions of messages/second over one million concurrent, connected devices can be supported on each MessageSight appliance. This enables a massive fan out streaming of data, processing 13 million non-persistent messages/second. When assured delivery matters, the device can process 400 thousand persistent messages/second.

  With MessageSight, message latency is measured in microseconds and remains consistent when scaled out. In this way, connecting many MQTT clients to MessageSight will not affect the internal latency of MessageSight.

- **Enabling integration with enterprises**
  MessageSight supports Java Message Service (JMS), MQTT over HTML5 websockets, and the MQTT protocol. It has built-in connectivity with IBM WebSphere MQ, making it possible to connect to multiple back-end queue managers.

**Note:** MessageSight can be used with all supported WebSphere MQ platforms WebSphere MQ version 7.1 and later.

### 3.1.1 MessageSight is a developer-friendly solution

With MessageSight, the simple yet powerful application programming interfaces (APIs) provided by MQTT clients make application development easy. A simple paradigm of connect/disconnect, subscribe/unsubscribe, and publish promotes loosely coupled and scalable applications.

Because of the efficiency of the MQTT messaging protocol, which is faster and requires less bandwidth and less battery power than traditional HTTP, MessageSight is optimized for wireless clients. The event-oriented paradigm provides for a better customer experience. The developer-friendly application programming interfaces (APIs) and libraries can be used for a variety of mobile OSs, including Google’s Android and Apple’s iOS.

**Note:** Developers can use a version of the MessageSight virtual appliance that is available at no charge from the IBM Mobile and machine-to-machine (M2M) community:

3.1.2 Connections to MessageSight

This section presents an overview of the components and connections in a MessageSight solution.

Figure 1-5 on page 14 shows examples of how clients who are connected to MessageSight can interface with WebSphere MQ and other back-end applications.

MessageSight supports client applications using specific protocols. At the time of publishing, IBM provides the following clients for these protocols:

- MQTT over TCP/IP:
  - MQTT C client
  - MQTT client for Android
  - MQTT client for Java
  - MQTT client for iOS

- MQTT over websockets
  - MQTT client for JavaScript

- JMS
  - MessageSight JMS client

3.2 Messaging patterns of MessageSight

Messaging patterns identify the common message flows that are used in messaging solutions. There are five messaging patterns that are supported by MessageSight:

- Fan out broadcast
- Fan in per device notification
- Fan out per device notification
- Fan out per device request-reply
- Fan in per device request-reply

3.2.1 Fan out broadcast

For this messaging pattern, one publisher device publishes a message to a specific topic string. The messages have many subscriber devices. Figure 3-2 on page 51 depicts the fan out broadcast pattern.
Figure 3-2  Fan out broadcast messaging pattern

One use of this pattern is when broadcasting data related to the location of a connected vehicle. As an example, this is useful for broadcasting an updated vehicle position.

### 3.2.2 Fan in per device notification

For this messaging pattern, many publisher devices publish messages to a topic string. The messages have one subscriber device. Figure 3-3 presents the fan in per device notification pattern.

Figure 3-3  Fan in per device notification messaging pattern

One use of this pattern is when receiving data from a number of sensors. As an example, this is useful when receiving data from earthquake sensors.
3.2.3 Fan out per device notification

For this messaging pattern, one publisher device publishes messages to many topic strings. Each message has only one subscriber device. Figure 3-4 presents the fan out per device notification pattern.

Figure 3-4  Fan out per device notification messaging pattern

One use of this pattern is when sending control commands to a device. As an example, this is useful when sending a command to an application to activate a feature.

For this messaging pattern, each subscriber must subscribe to a unique topic. By using the unique client ID, group ID, or user ID of the subscribing application in the topic string, you can ensure that each topic is unique. For example, a subscriber application instance that connects with client ID 123 subscribes to RESPONSE/123. A subscriber application instance that connects with client ID 456 subscribes to RESPONSE/456. A subscriber application instance with client ID 789 subscribes to RESPONSE/789.

To ensure that subscribers cannot access topics of other subscribers, you can use the topic string variable substitution that is available in MessageSight messaging policies. By using topic string variable substitution, you can create a messaging policy with a single topic string that includes a user ID, group ID, or client ID variable.

This substitution ensures that applications can subscribe only to the topic string that matches their user ID, group ID, or client ID. For example, a topic string of RESPONSE/${ClientID} is specified in the messaging policy. The application with client ID 123 is allowed to subscribe to RESPONSE/123, but is not allowed to subscribe to RESPONSE/456 or RESPONSE/789 because the client IDs do not match.
3.2.4 Fan out per device request-reply

For this messaging pattern, one publisher device publishes messages to many topic strings. Each topic string has only one subscriber device. Each subscriber device publishes reply messages on a separate topic string. The publisher device subscribes to all the reply topics. Figure 3-5 presents the fan out per device request-reply pattern.

![Fan out per device request-reply messaging pattern](image)

One use of this pattern is when a control center is interrogating the state of a device. As an example, this is useful when requesting a temperature reading from a sensor.

3.2.5 Fan in per device request-reply

Many publisher devices publish messages to many topic strings. A single subscriber device subscribes to all of the topic strings. The subscriber device publishes reply messages on separate topic strings for each publisher device. Figure 3-6 on page 54 presents the fan in per device request-reply pattern.
One use of this pattern is when a device is polling a control center for information updates. As an example, this is useful when polling for information about firmware updates.

### 3.3 Install the MessageSight virtual appliance (for developers)

This section describes how the development community can install the MessageSight virtual appliance at no charge.

This virtual edition is to be used in a development environment only. The benefit of this developer-focused virtual appliance is that it enables rapid application development. It enables developers to get started quickly in coding and testing applications to work with MessageSight. This virtual appliance is suitable for deploying in IBM VMware and Oracle VirtualBox environments.

**Note:** All details regarding the installation of the MessageSight physical appliance installation are in the publication, *Responsive Mobile User Experience Using MQTT and IBM MessageSight*, SG24-8183.

**Note:** Developers can use the version of the MessageSight virtual appliance that is available at no charge from the IBM Mobile and M2M community:

After using the Open Virtual Archive (OVA) file of the MessageSight virtual edition, follow these steps for a VMware workstation environment:

1. If not already running, start VMware.
2. Select **File → Open** from the menu.
3. Import the OVA file by entering a name for the new virtual machine and a storage path.
4. Navigate to the location of the MessageSight OVA image.
5. Select the `IBMMessageSightVx.y.ova` file and click **Open**.

**Note:** At the time of publishing, the latest release of the MessageSight virtual appliance is 1.1. Therefore, the OVA file to import is `IBMMessageSightV1.1.ova`.

Wait for the end of the import process. The import process might take several minutes to complete.

6. Define the settings for the newly imported MessageSight virtual image. The minimum requirements to run the MessageSight virtual appliance are listed:
   - Memory: 4 GB
   - Processors: 2
   - Hard disk: 16 GB
   - Network settings: In this installation example, we use a network address translation (NAT) network adapter. The NAT is configured to use the dynamic host configuration protocol (DHCP), but you can choose any network adapter type. If you are not using DHCP, see the MessageSight Ethernet interfaces configuration guide available in the MessageSight IBM Knowledge Center:

7. Click the **Power on this virtual machine** link.

Wait for the virtual machine to start. This might take a couple of minutes.

Figure 3-7 shows the start of the boot process of the MessageSight virtual appliance.

```
>> Parsing ELF... OK. oting 'IBM MessageSight'
> Progress: ▼▼▼
```

**Figure 3-7**  The boot process begins on the MessageSight virtual appliance

8. Click in the VMware window.
9. Type `admin` as the login user ID and press Enter, as shown in Figure 3-8.

```
(none) login: admin
```

**Figure 3-8**  Log in as administrator to connect to the virtual MessageSight appliance

10. Type `admin` as the password and press Enter.
11. At this step, an Ethernet adapter must be configured. Press Enter to accept the default adapter: `eth0`, as shown in Figure 3-9 on page 56.
12. Enter yes and press Enter to select DHCP rather than assigning a fixed address.

13. Press Enter to confirm the selection, as shown in Figure 3-10.

14. The interface is configured using DHCP. Record the assigned address. This address is used to communicate with the appliance.

The command `ethernet-interface eth0` can be used to display the address assigned to an Ethernet port, as shown in Figure 3-11.
15. To exit the VMware image, press Ctrl + Alt simultaneously.

**Note:** If you make a mistake in configuring the value of `eth0`, type the following command at a command prompt and make the necessary corrections:

```
edit ethernet-interface eth0
```

### 3.4 Overview of the MessageSight web UI

This section presents an overview of the web UI for the physical and virtual MessageSight appliance. Access the UI from a supported web browser. For a list of supported web browsers, visit the MessageSight IBM Knowledge Center:


**Note:** All details regarding the installation of the MessageSight physical appliance installation are in the Redbooks publication, *Responsive Mobile User Experience Using MQTT and IBM MessageSight*, SG24-8183.

The host name or IP address of the MessageSight appliance is required to access the web UI. The HTTPS protocol is used to secure the communication between appliance users and the MessageSight appliance.

**Note:** The default listening port of the MessageSight web UI is 9087. This value is configurable.

### 3.4.1 Connect to the MessageSight appliance

The MessageSight certificate that establishes secure communications with the web browser is a self-signed certificate. Therefore, your browser might warn you that the connection is untrusted. Confirm the security exception to access the MessageSight web UI.

Use the following steps to connect to the MessageSight appliance using the web UI:

1. Open a supported web browser and enter the following URL:

   `https://<MessageSight_HostName_Or_IPAddress>:9087`

2. Accept the MessageSight self-signed certificate, and the login window shown in Figure 3-12 on page 58 opens.
3. Enter the login and password to access the MessageSight web UI. The default values are shown:
   - Login: admin
   - Password: admin

4. View and accept the software licensing agreement that displays at your first connection to the web UI.

   The MessageSight web UI First Steps tab displays. Here, you can configure the range for Ethernet client connections, and configure a default gateway, or change the default password for the administrator account, as shown in Figure 3-13 on page 59. To change the default administrator password at a later time, see “Reset an existing password (all user roles)” on page 69.
Figure 3-13  MessageSight web UI First Steps tab

**Note:** Use the Classless Inter-Domain Routing (CIDR) notation to define the IP address range property.

5. Click **Save and Close** to save and close your changes, and click the **Home** tab to display the MessageSight web UI home page, as shown in Figure 3-14 on page 60.
3.4.2 The MessageSight Home page

The MessageSight home page is divided into two sections:

- The *Common configuration and customization tasks* section is shown in Figure 3-15 on page 61.
This panel guides you through the configuration and test of the following elements:

- **Verify your network configuration with the MessageSight Messaging Tester**

  The Messaging Tester is a simple graphical tool that can help to test the connections to a network interface of a MessageSight appliance. The Messaging Tester is a web client based on JavaScript, which uses MQTT over websockets for communications with MessageSight through one of its network interface adapters.

  Using the Messaging Tester, you can configure up to five MQTT clients that are executing publications and subscriptions, as shown in Figure 3-16 on page 62.
– Customize appliance settings
  Here, you can configure the network, date, and time settings, and perform other system tasks.

– Secure your appliance
  Security profiles, groups, and messaging users are configured through the Security Settings. The configuration parameters of the web UI are also accessible through this menu.

– Create users and groups
  Two types of users can be defined and a MessageSight appliance: the appliance users (people who need to connect to MessageSight to configure or monitor the appliance) and the messaging users (who access message hubs and endpoints configured on the appliance through MQTT), and JMS clients (a mobile device user, for instance).

– Configure MessageSight to accept connections
  This menu is used to configure message hubs, endpoints, messaging and connection policies, and MQ Connectivity.

The Appliance Dashboard section is shown in Figure 3-17 on page 63.
Figure 3-17  MessageSight appliance web UI dashboard

The appliance dashboard displays the following information:

- **Quick Stats**: The numbers of accepted and rejected incoming messages, active connections, messages per second, appliance uptime, disk and memory usage, and last firmware update.

- **Active connections and Throughput**: Shows the average active connections and average messages per second.

The following sections describe the execution of several basic administration tasks using the MessageSight web UI.

### 3.4.3 Administrator actions using the MessageSight web UI

Several of the more common administrator tasks using the web UI are discussed in this section.

**Determine the status of the MessageSight server**

The status of a MessageSight server can be determined by using the Status selection list that displays at the top of the MessageSight control panel, as shown in Figure 3-18 on page 64.
In our example, the server is running, and ready for use.

**Display the firmware version of a MessageSight appliance**

There are two ways to display the firmware version from the web UI. The first method is used by all appliance users. The second method can be used by administrators only.

- **First method**
  
  All appliance users can use this method. For a description of the user roles available in MessageSight, see “User roles and configuring users” on page 66.
  
  Connect to MessageSight using the web UI, click the Help (?) icon in the menu bar, as shown in Figure 3-19, and select **About**.

  ![Figure 3-19 Display the firmware version using the About menu](image)

  A window displays the firmware version and build number, as shown in Figure 3-20 on page 65.
Second method

Only MessageSight administrators can use this method for displaying the firmware version. For a description of all user roles available in MessageSight, see “User roles and configuring users” on page 66.

Connect to MessageSight using the web UI, and select **Appliance → System Control**, as shown in Figure 3-21 on page 66.
The firmware version displays, as shown in Figure 3-22.

User roles and configuring users
MessageSight supports role-based user actions. The user roles are listed:

- System administrator: Users who have access to every task on the appliance.
- Messaging administrator: Users who have access only for viewing or editing messaging-related tasks. For example, a messaging administrator can configure an endpoint but is not able to configure the SSL or TLS certificates.
- Appliance users: Users who have access to the MessageSight messaging-related views.

For further details about the actions that each type of appliance user can perform, see the following URL:

System administrators can access MessageSight using the web UI or CLI. Access to the CLI interface using Secure Shell (SSH) is denied for messaging uses and appliance users, as shown in Figure 3-23.

```
login as msgadmin
Using keyboard-interactive authentication.
Password:
Using keyboard-interactive authentication.
You must be a member of cn=SystemAdministrators,ou=groups,ou=webui,dc=ism,ibm,dc=com to login.
Access denied
```

Figure 3-23  Access is denied to the MessageSight CLI for users other than system administrators

**Configure users**

To add, edit, or delete appliance users, complete these steps. This process can be performed only by a system administrator:

1. Connect to the MessageSight web UI using the administrator account. The Home page opens.
2. Under Create users and groups, select **Appliance Users**, as shown in Figure 3-24.

![Create appliance users from the web UI Home page](image)

Figure 3-24  Create appliance users from the web UI Home page

The list of existing appliance users displays, as shown in Figure 3-25 on page 68.
Other ways to add, edit, or delete users, including resetting passwords, are shown:

- **Add a user:**
  
  Click the **plus sign (+)** icon. In the window that opens, enter a user name and password, and select a group name to associate the user with: *SystemAdministrators*, *Users*, or *MessagingAdministrators*.

- **Edit a user password:**
  
  Select the user to edit and click the **pencil (edit)** icon.

- **Delete a user:**
  
  Select the user to delete and click the **delete (red cross)** icon.

**Add an appliance user**

To create a new appliance user ID, follow these steps:

1. Navigate to the Appliance Users page and click the **plus sign (+)** icon.
2. In the web UI Users section, click the **plus sign (+)** icon.
3. When the Add User pop-up box displays, provide the User ID, Password, and Description and select the type of user, as shown in Figure 3-26 on page 69.
4. Populate the required details, and click **Save**.

**Reset an existing password (all user roles)**

To reset a password on an account for any user role, highlight the account and select **Other Actions → Reset Password**, as shown in Figure 3-27, which is an example of changing the password for the messaging administrator. Note that this procedure can be used to change the initial, default administrator password, as well.

---

**Figure 3-26**  Adding a new user in the web UI

**Figure 3-27**  Password reset for the messaging administrator ID
Reset an administrator password (alternate method)

Alternatively, the administrator account password can be changed by navigating to the admin user menu that displays at the upper right of the MessageSight web UI, as shown in Figure 3-28.

![Figure 3-28  Change the administrator password from the admin user menu](image)

Any user who is logged in to MessageSight through the web UI can change their password using the Change Password link that displays in the user menu for that user.

3.5 Overview of the MessageSight CLI

The MessageSight appliance provides an extensive CLI, which includes a few utilities and diagnostic tools that are not available using the web UI.

Note: All details related to the installation of the MessageSight physical appliance are in the Redbooks publication, Responsive Mobile User Experience Using MQTT and IBM MessageSight, SG24-8183.

If you have installed a physical appliance, you can access the MessageSight CLI locally using a keyboard, video, and mouse (KVM) console, or over the network using serial over local area network (LAN) or SSH.

If you have installed a virtual MessageSight appliance, you can access that CLI directly from your hypervisor solution, if you have an administrator name and password, or by using SSH, as well.

For this discussion, we access the MessageSight appliance CLI using an SSH client.

3.5.1 Connect to the MessageSight appliance

Complete these steps to access your MessageSight appliance using SSH:

1. Open your SSH client.
2. Connect to MessageSight using the appliance host name or IP address and port 22, which is the connection port reserved for SSH.
3. At connection time, accept the host key presented by the MessageSight server at the first connection attempt because this host key is not cached in the SSH client registry at the moment.
4. Enter your system administrator login and password to connect to the CLI.

When you are connected, the CLI console displays the response shown in Example 3-1 on page 71.
Example 3-1 Accessing the CLI console using SSH

login as: admin
Using keyboard-interactive authentication.
Password:
Last login: Wed Jul 9 03:24:00 UTC 2014 from 192.168.198.1 on pts/0
Welcome to MessageSight
5725-F96
Copyright 2012, 2013 IBM Corp. Licensed Materials - Property of IBM.
IBM and MessageSight are trademarks or registered trademarks of IBM, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies.
Console>

5. Enter the necessary CLI commands to administer, configure, or monitor your MessageSight appliance.

Note: Access the MessageSight V1.1 Command Reference guide in the MessageSight IBM Knowledge Center website:

3.5.2 Administrator actions using the MessageSight CLI

In this section, we introduce several of the administrator commands for use with the MessageSight CLI.

Determine the machine type and serial number of a physical or virtual MessageSight appliance

To determine the machine type and serial number of your physical appliance, use the show version command, as shown in Example 3-2.

Example 3-2 Determine your machine type and serial number using the CLI

Console>
Console> show version
Installation date: Dec 4, 2013 8:54:50 PM
Platform version: 5.0.0.19
Platform build ID: build22-20131007-1001
Platform build date: 2013-10-07 14:08:05+00:00
Machine type/model: 7915AC1
Serial number: KQ2H7YC
Entitlement: KQ2H7YC
Firmware type: Release
Console>

You might notice a difference for the Machine type/mode output value if you apply the same show version command on a MessageSight virtual appliance. This is shown in Example 3-3 on page 72.
Example 3-3  Machine type and serial number on a MessageSight virtual appliance

Console>
Console> show version
Installation date: Nov 12, 2013 4:34:04 AM
Platform version: 5.0.0.19
Platform build ID: build22-20131007-1001
Platform build date: 2013-10-07 14:08:05+00:00
Machine type/model: VMware Virtual Platform
Serial number: VMware-56 4d 62 8e 75 24 93 50-32 c0 10 e8 eb 83 fc 5a
Entitlement: VMware-56 4d 62 8e 75 24 93 50-32 c0 10 e8 eb 83 fc 5a
Firmware type: Release
Console>

Determine the status of the Ethernet interface

Use the MessageSight CLI to get the status of an Ethernet interface using the `status ethernet-interface` command, as shown in Example 3-4, where `eth0` is the network interface of the appliance for which you want to retrieve a status.

Example 3-4  Determine the status of an Ethernet interface using the CLI

Console>
Console> status ethernet-interface eth0
eth0    OpState:[Up]
generic MTU:1500 carrier:true
    flags:UP BROADCAST RUNNING MULTICAST index:5
inet addr:192.168.198.40 flags:PERMANENT mask:255.255.255.0
    scope:GLOBAL
inet6 addr: fe80::20c:29ff:fe83:fc5a flags:PERMANENT
    mask: ffff:ffff:ffff:ffff:: scope:LINK
ethernet Link:on MAC: 00:0c:29:83:fc:5a autoneg:on duplex:Full
    port:TP speed:1000Mbps
statistics collisions:0 multicast:0 rx_bytes:2330429
    rx_compressed:0 rx_crc_errors:0 rx_dropped:0 rx_errors:0
    rx_fifo_errors:0 rx_frame_errors:0 rx_length_errors:0
    rx_missed_errors:0 rx_over_errors:0 rx_packets:22383
    tx_aborted_errors:0 tx_bytes:48742 tx_carrier_errors:0
    tx_compressed:0 tx_dropped:0 tx_errors:0 tx_fifo_errors:0
    tx_heartbeat_errors:0 tx_packets:354 tx_window_errors:0
Console>

Set and get the node name of a MessageSight appliance

Use the MessageSight CLI to set the appliance node name using the `nodename set` command, as shown in Example 3-5, where `imaDev` is the appliance node name.

Example 3-5  Set an appliance node name using the CLI

Console>
Console> nodename set imaDev
Console>

To determine the node name of an appliance, use the `nodename get` command, as shown in Example 3-6 on page 73.
Example 3-6  Determine an appliance node name using the CLI

```
Console> nodename get
nodename is imaDev
Console>
```

Display the firmware version of a MessageSight appliance

Use the MessageSight CLI to determine the firmware version of an appliance using the `show imaserver` command, as shown in Example 3-7.

Example 3-7  Determine the firmware version of a MessageSight appliance using the CLI

```
Console> show imaserver
MessageSight version is 1.1 20131112-0402 2013-11-11 23:04
Console>
```

Gather information for use by the MessageSight support team

Use the MessageSight CLI to gather information about an appliance using the `platform must-gather` command. This information is useful for the MessageSight support team. The web UI cannot be used to gather this information.

The `platform must-gather` command is used for troubleshooting, and for collecting the must-gather diagnostic data for reporting a problem to the MessageSight support team when generating a problem management record (PMR). Sending the output from the `platform must-gather` command helps the MessageSight support team to resolve your problem more quickly.

To collect this output, complete the following steps:

1. Access the MessageSight CLI.
2. Run the `platform must-gather` command to create a .tgz file containing diagnostic information. If you have a PMR open with IBM, include that number in the file name.

   ```
   Console> platform must-gather PMR-12345,67R,890-Jul16.tgz
   ```
3. Run the `file list` command. You will see the file that you created, and a separate `collect-pd.txt` file, as shown in Example 3-8.

   ```
   Example 3-8  Listing files
   Console> file list
   Console> file list
   collect-pd.txt 726876 bytes created Jul 9, 2014 4:25:21 AM
   Console>
   ```
4. Send the output from the `platform must-gather` command to the MessageSight support team as part of a PMR problem ticket.

Create a secure backup of a MessageSight appliance

Use the MessageSight CLI to create a secure backup of an appliance using the `imaserver backup` command. The web UI cannot be used for this task.
For more information about the MessageSight backup and restore processes, see the MessageSight IBM Knowledge Center:


To create a secure backup of an appliance, complete the following steps:

1. Check that the server is running by entering the `status imaserver` command, as shown in Example 3-9.

   **Example 3-9  Determine the status of a MessageSight appliance**

   ```
   Console>
   Console> status imaserver
   Status = Running (production)
   ServerUpTime = 0 days 9 hours 59 minutes 26 seconds
   Console>
   ```

   If the server is stopped, start the server by entering the following command:

   ```
   Console> imaserver start
   ```

2. Set the server to maintenance mode by entering the commands shown in Example 3-10.

   **Note:** Maintenance mode is a MessageSight server state that allows a system administrator to complete several tasks. In maintenance mode, a system administrator can clean the server store, stop and start the server, and change the server state from maintenance to production.

   **Example 3-10  Set the MessageSight appliance to maintenance mode**

   ```
   Console>
   Console> imaserver runmode maintenance
   The MessageSight server is currently in "production" mode.
   When it is restarted, it will be in "maintenance" mode.
   Console> imaserver stop
   The MessageSight server is stopping.
   Check the MessageSight server status using "status imaserver" command.
   Console> imaserver start
   The MessageSight server is starting.
   The MessageSight server is in "maintenance" mode.
   Console>
   ```

3. Create a secure backup of the appliance by using the `imaserver backup` command:

   ```
   Console> imaserver backup "Password=mybackup_password"
   ```

   **Note:** The `mybackup_password` command specifies a password for the compressed file that contains the backup data. You must have the password when you attempt to restore the backup.

4. Set the server to production mode, by entering the commands shown in Example 3-11 on page 75.
Example 3-11  Set the MessageSight server to production mode

Console>
Console> imaserver runmode production
The MessageSight server is currently in "maintenance" mode. When it is restarted, it will be in "production" mode.
Console> imaserver stop
The MessageSight server is stopping. Check the MessageSight server status using "status imaserver" command.
Console> imaserver start
The MessageSight server is starting. The MessageSight server is in "production" mode.
Console>

Note: Information about how to restore a MessageSight appliance is in the MessageSight IBM Knowledge Center:

- Restoring the MessageSight configuration data to the same appliance:
- Restoring the MessageSight configuration data to a different appliance:

3.6 Message hubs, endpoints, and policies

Message hubs are an organizational configuration object that collects the following components:

- Endpoints
- Connection policies
- Messaging policies

These components are associated with a specific goal in a single place. You can create a message hub for an application to organize the endpoints and policies that each application uses.

3.6.1 Endpoints

Endpoints accept network requests so that clients can connect to the message hub through MQTT or JMS. You must have a minimum of one endpoint per message hub. You can create one endpoint for each port the message hub listens on.

Endpoints must have one or more connection policies that are applied to them to accept client connections. Endpoints must have at least one messaging policy that is applied to them for a connected client to be able to publish/subscribe.
Endpoints can be used only within the specified message hub. Separate message hubs cannot specify endpoints on the same IP address and ports. Different message hubs can have endpoints with different IP addresses and the same port. This architecture means that endpoints can be monitored to provide a complete picture of traffic and activity across the message hub. For example, endpoints can be used to track connections and monitor metrics, such as the range of IP addresses used to connect to the message hub, the number of incoming messages, and the number of subscriptions that are associated with a particular message hub.

There are two types of policies that can be defined on the message hub:

- **Connection policies**: A connection policy filters any connection that is based on a rule. For example, a connection policy might be set up to authorize users that are defined within a particular group or groups to connect from a specific location or range of locations.

- **Messaging policies**: A messaging policy authorizes a client or user to publish or subscribe to a topic, or to send, receive, or browse messages on a queue. You can apply your messaging policy to a topic or to a queue as part of the specification of the message policy.

**Note**: Messaging policies that are created in the web UI can be used only within the specified message hub. Messaging policies that are created in the CLI can be attached to any endpoint in any message hub.

### 3.6.2 Message hubs

A message hub is an organizational object that groups endpoints, connection policies, and messaging policies that are associated with a specific goal. For example, you can create a message hub per application to organize the endpoints and policies that each application uses.

Figure 3-29 presents a simplified view of a message hub with its components and cardinalities between components of a specific message hub.
### 3.6.3 Connection policies

A *connection policy* is used to authorize a client to connect to an endpoint. The connection policy can restrict which clients can connect to the endpoint. You must apply at least one connection policy to an endpoint so that a client can connect to that endpoint. When you create a connection policy, you can use the following filter attributes to restrict who is allowed to connect:

- Client IP address
- Client ID
- User ID
- Group Name
- Protocol
- Certificate common name

A connection policy can be applied to more than one endpoint that is defined in the same message hub. For example, you can use a single connection policy to allow all clients from a particular IP address range to connect. You can then restrict the access of different clients to particular queues and topic strings by using a messaging policy.

### 3.6.4 Messaging policies

A *messaging policy* is used to control the topics or queues for which a client can send and receive messages. When you create a messaging policy, you must specify the following components:

- Name
- Destination Type: Topic, Global-shared subscription, or Queue
- Destination: Value of the topic
- Max Messages (only valid for topics)
- Authority: Publish, Subscribe, Send, Browse, Receive, or Control

You must specify at least one of the following filters:

- Client IP address
- Client ID
- User ID
- Group Name
- Certificate Common Name
- Protocol: JMS or MQTT

**Note:** When specifying the Destination, you can use an asterisk (*) to specify all topic strings or queues. You can also use variable substitution in the topic string or queue to ensure that only specific user IDs, group IDs, client IDs, or client certificate common names can access a topic. The variable for the user ID is `${UserID}`.

The substitution variables are listed:

- `${UserID}` for the user ID
- `${GroupID}` for the group ID
- `${ClientID}` for the client ID
- `${CommonName}` for the client certificate common name

For example, if a topic string in a messaging policy is `Pickmeup/drivers/${ClientID}`, a client with an ID of `driver_a` can access the topic `Pickmeup/drivers/driver_a`. A client with an ID of `driver_b` cannot access the topic `Pickmeup/drivers/driver_a`, but that client can access `Pickmeup/drivers/driver_b`. 

---

**Chapter 3. Overview of IBM MessageSight**

77
3.6.5 Endpoints

An endpoint enables a client to connect to the MessageSight appliance. As shown in Figure 5-2 on page 107, each endpoint must have at least one connection policy, and at least one messaging policy.

When you create an endpoint, you can specify the following attributes:

- Name
- Enabled: True or False
- IP address
- Port
- Protocol: JMS or MQTT
- Maximum message size
- Security profile
- Connection policies
- Messaging policies

3.6.6 The DemoHub message hub

Every virtual or physical MessageSight appliance automatically integrates a DemoHub message hub. This message hub includes an endpoint (DemoMqttEndpoint) that listens on Port 1883, without any security constraints.

The purpose of this message hub is to let developers rapidly integrate an application after MessageSight is installed. The MessageSight Messaging Tester also uses this message hub to test a network interface. See “3.4, "Overview of the MessageSight web UI" on page 57.

The DemoHub message hub has the following characteristics:

1. Connect to your MessageSight appliance using the web UI.
2. From the top-level menu, select **Messaging → Message Hubs** to access the Message Hubs configuration page, as shown in Figure 3-30.

![Figure 3-30 Accessing the Message Hubs configuration page](image)

3. In the list of configured message hubs, you can see the DemoHub message hub, as shown in Figure 3-31 on page 79.
Chapter 3. Overview of IBM MessageSight

4. Select DemoHub and click the pencil (edit) icon, as shown in Figure 3-32.

5. Select the Endpoints tab as shown in Figure 3-33.

6. Select DemoMqttEndpoint and click the pencil (edit) icon to access its configuration, as shown in Figure 3-34 on page 80.
A new window opens, showing the configuration properties of the DemoMqttEndpoint endpoint, as shown in Figure 3-35.

Notice the listening port of the endpoint (1883) and the connection and messaging policies. This endpoint does not provide any security control and it is configured to support the MQTT protocol only.

7. Click Close to close the opening window.

8. Select the Messaging Polices tab from the DemoHub panel, as shown in Figure 3-36 on page 81.
9. Select the **DemoMessagingPolicy** messaging policy and click the **pencil (edit)** icon.

10. A window opens, showing the configuration parameters of the DemoMessagingPolicy messaging policy, as shown in Figure 3-37.

Notice the Destination (*) and the Authority (Publish and Subscribe) properties, which indicate that publication and subscription can be done on any topic. JMS and MQTT protocols have been selected, as well. Therefore, the defined messaging policy allows access when the protocol is JMS or MQTT.
3.6.7 Configuring your first message hub using the MessageSight web UI

The creation of a message hub using the web UI is described in 5.2.1, “MessageSight basic configuration” on page 109.

3.6.8 Configuring a message hub using the MessageSight CLI

The process to configure a message hub using the MessageSight CLI is described.

List of commands to configure a message hub
This section describes the CLI commands for use in configuring message hubs, including their components (endpoints, connection policies, and messaging policies).

To configure a message hub and its components using the CLI, it is important to adhere to the following order at creation time:
1. Message hubs
2. Connection policies
3. Messaging policies
4. Endpoints

The commands used to configure a message hub using the CLI are shown in Example 3-12.

Example 3-12 CLI commands used to configure a Message Hub

```
imaserver create
imaserver delete
imaserver list
imaserver show
imaserver update
```

Note: For details about the commands specific to message hubs, see Message hub commands in the MessageSight IBM Knowledge Center:

Create a basic message hub using the CLI
In this section, we describe the creation of a basic message hub using the CLI commands.

The properties of the message hub that we create in this section are the same as those of the DemoHub message hub, which was presented in 3.6.6, “The DemoHub message hub” on page 78, with a few differences. The differences are the names of the hub components:

- The name of the message hub we create using the CLI is PickmeupHub.
- The name of the endpoint of the PickMeUpHub is PickmeupEndpoint.
- The name of the connection policy associated with the PickMeUpEndpoint is PickmeupConnPolicy.
- The name of the messaging policy associated with the PickMeUpEndpoint is PickmeupMsgPolicy.
The sequence of CLI commands for creating the PickMeUp message hub and its components are shown in Example 3-13.

**Example 3-13  CLI commands to create the PickMeUp message hub and its components**

```
Console> imaserver create MessageHub "Name=PickmeupHub" "Description=message hub for the PickmeUp application"
The requested configuration change has completed successfully.
Console>
Console> imaserver create ConnectionPolicy "Name=PickmeupConnPolicy" "Description=connection policy for the PickmeUp app" "Protocol=MQTT"
The requested configuration change has completed successfully.
Console>
Console> imaserver create MessagingPolicy "Name=PickmeupMsgPolicy" "Description=messaging policy of the PickmeUp app" "DestinationType=Topic" "Destination=" "MaxMessages=5000" "ActionList=Publish,Subscribe" "DisconnectedClientNotification=False" "Protocol=MQTT"
The requested configuration change has completed successfully.
Console>
Console> imaserver create Endpoint "Name=PickmeupEndpoint" "Description=endpoint of Pickmeup message hub" "Port=16000" "Interface=all" "Protocol=MQTT" "ConnectionPolicies=PickmeupConnPolicy" "MessagingPolicies=PickmeupMsgPolicy" "MessageHub=PickmeupHub" "Enabled=True"
The requested configuration change has completed successfully.
```

The last command in Example 3-13 is `imaserver list MessageHub`, which is used to display the list of message hubs that are configured on a MessageSight appliance. As you can see, PickmeupHub is part of the list.

Connect the MessageSight web UI to confirm that the PickMeUp message hub has been created, as shown in Figure 3-38.

![Figure 3-38  List of message hubs](image)
Retrieve configuration information

The `imaserver show` command is used to retrieve the configuration properties of a message hub, a messaging policy, a connection policy, or an endpoint.

In Example 3-14, the `imaserver show` command displays the configuration information of the PickmeupEndpoint endpoint.

**Example 3-14   CLI command to show endpoint information**

```plaintext
Console>
imaserver show Endpoint "Name=PickmeupEndpoint"
Name = PickmeupEndpoint
Enabled = True
Port = 16000
Protocol = MQTT
Interface = all
SecurityProfile =
ConnectionPolicies = PickmeupConnPolicy
MessagingPolicies = PickmeupMsgPolicy
MaxMessageSize = 1024KB
MessageHub = PickmeupHub
Description = endpoint of Pickmeup message hub
Console>
```

### 3.6.9 Use the MessageSight SSH to deploy message hub configuration

In this section, we present a solution to deploy the configuration of a message hub on a MessageSight appliance using the MessageSight SSH service.

The `bash` script is shown in Example 3-15. The idea is to make the creation of message hubs easier using a deployment script. This deployment script can be created, based on an XML or JavaScript Object Notation (JSON) parameter file that can be implemented by a messaging administrator.

**Example 3-15   Bash script example for deploying a MessageHub configuration**

```bash
#!/usr/bin/bash
#
# This script automates an SSH session with a MessageSight appliance.
# In this session, it creates a MessageHub including a Connection policy + a Messaging policy + an Endpoint.
# ARGSS=2
if [ $# -ne "$ARGS" ]; then
  echo "Usage: ima-createHub.sh USER IMA_HOSTNAME"
  exit 1
```
fi

ima_user=$1
ima_hostname=$2
commands_file="ima_commands.txt"
commands_execution_output="ima_commands_execution_output.txt"

# remove old files to make sure we create new files, instead # of appending to old ones
rm -f $commands_file
rm -f $commands_execution_output

#echo ima_hostname=$ima_hostname

# create list of commands, in a file. These commands # will be sent to the MessageSight ssh service
echo imaserver create MessageHub "\"Name=MyMessageHub\" \"Description=message hub created using CLI commands\"" >> $commands_file
echo imaserver create ConnectionPolicy "\"Name=MyConnPolicy\" \"Description=connection policy of MyMessageHub\" \"Protocol=MQTT\"" >> $commands_file
echo imaserver create MessagingPolicy "\"Name=MyMsgPolicy\" \"Description=messaging policy of MyMessageHub\" \"DestinationType=Topic\" \"Destination=myorg/sample\" \"MaxMessages=5000\" \"ActionList=Publish,Subscribe\" \"DisconnectedClientNotification=False\" \"Protocol=MQTT\"" >> $commands_file
echo imaserver create Endpoint "\"Name=MyEndpoint\" \"Description=endpoint of MyMessageHub\" \"Port=16003\" \"Interface=all\" \"Protocol=MQTT\" \"ConnectionPolicies=MyConnPolicy\" \"MessagingPolicies=MyMsgPolicy\" \"MessageHub=MyMessageHub\" \"Enabled=True\"" >> $commands_file
#echo imaserver list MessageHub >> $commands_file
echo exit >> $commands_file
chmod 400 $commands_file

# redirect the output of the ssh session to a file, so we # can grep it and see how many commands were successfully executed by MessageSight
ssh -l $ima_user $ima_hostname < $commands_file 2> $commands_execution_output

any_failure=`grep -E 'required|not valid|invalid' $commands_execution_output`

# remove the commands file after using it
#rm -f $commands_file

echo "*** MessageSight Configuration Status ***"
if [ "$any_failure" ]; then
  echo "Failure occurred creating MessageSight configuration"
  exit 1
else
  echo "MessageSight configuration created successfully"
  exit 0
fi

The complete deployment process is shown in Figure 3-39 on page 86.
Figure 3-39  Using the MessageSight SSH service to deploy a message hub configuration
Typical network topology, messaging patterns, and considerations

Network topology is the arrangement of the various elements of a computer network. It describes the placement of the various components in various physical domains and the interactions among those components. This chapter provides an overview of the typical network topologies, messaging patterns, and considerations commonly implemented with IBM MessageSight.

This chapter includes the following topics:

- 4.1, “Network topology” on page 88
- 4.2, “Messaging patterns” on page 91
- 4.3, “Messaging considerations” on page 102
4.1 Network topology

The network topology of a machine-to-machine (M2M) system varies depending on the type and number of devices being connected. The following list shows important factors that influence this decision:

- **Latency**: The time it takes for a packet of data to be transmitted from the sensor node through the network to the gateway node, or vice versa, from the gateway to the sensor node. Latency is directly proportional to the speed of the network: the faster the network, the lower the latency.

- **Throughput**: The amount of data to be transmitted through the network per second. Relatively high throughput is required for dense and complex data.

- **Fault resiliency**: The speed at which a network, if interrupted, will recover and deliver a packet of data to its destination.

- **Scalability**: The number of devices or nodes that can be included in a single network.

- **Hops**: The transmission of a data packet from one node to another. Hops refers to the number of nodes through which a data packet travels.

- **Range**: The range of the network is the overall distance a complete network can span. The range of a node is the maximum distance of one hop, from one node to the other.

To handle these challenges, a variety of topologies can be used. However, to accelerate the design and implementation, the selected topology needs to be built on standard connectivity patterns and components, to the greatest extent possible. The basic topology patterns that are most relevant to the discussion of IBM MessageSight and the MQTT protocol are shown in Figure 4-1.

![Figure 4-1 Client to edge server to enterprise server](image-url)
The MQTT connectivity solution works well in these topologies. It provides a lightweight messaging option that can be critical if the network between the devices and the central system components is constrained due to latency, limited bandwidth, or high incremental costs. And because the MQTT client libraries are small and require limited processing capacity on the devices that host them, the solution can work even with constrained devices.

This topology connects the devices to the enterprise integration layer through the edge gateway. The edge gateway acts a hub or concentrator for devices to connect to, and transports messages between the devices and enterprise integration layer. This topology is ideal for systems with an unreliable or insecure network, or where the number of devices can cause a performance slowdown if they are connected directly to the server in the enterprise integration layer.

Table 4-1 lists the components of a client to edge to enterprise server topology.

<table>
<thead>
<tr>
<th>Components</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business application</td>
<td>An application that sends and receives messages to or from a device. It connects to the server in the enterprise integration layer using a supported transport protocol, such as Java Message Service (JMS), WebSphere MQ, or MQTT.</td>
</tr>
<tr>
<td>Enterprise integration layer</td>
<td>The server here acts as a central concentrator that remote devices connect to. The following list reflects other activities that the server supports:</td>
</tr>
<tr>
<td></td>
<td>▶ Acts as an enterprise service bus for business applications’ message exchange.</td>
</tr>
<tr>
<td></td>
<td>▶ Capable of storing events and notifications for later delivery, if connectivity was not available when first attempted.</td>
</tr>
<tr>
<td></td>
<td>▶ Capable of enforcing connection security by means of Transport Layer Security (TSL)/Secure Sockets Layer (SSL) to authenticate remote devices.</td>
</tr>
<tr>
<td></td>
<td>▶ Capable of mediation for filtering and aggregating messages, such as to reduce the volume of messages traveling over the network.</td>
</tr>
<tr>
<td></td>
<td>▶ Capable of event correlation, such as merging or mapping an event or alarm between the devices and business applications.</td>
</tr>
<tr>
<td>Edge gateway</td>
<td>Serves as a hub or concentrator for devices to connect; supports the following functions:</td>
</tr>
<tr>
<td></td>
<td>▶ Acts as a gateway to and from the enterprise integration layer using a single connection.</td>
</tr>
<tr>
<td></td>
<td>▶ Allows devices to communicate with each other without going through the enterprise integration layer.</td>
</tr>
<tr>
<td></td>
<td>▶ Capable of storing events and notifications for later delivery, if connectivity was not available when first attempted.</td>
</tr>
<tr>
<td></td>
<td>▶ Capable of applying application logic, such as filtering or aggregation of messages.</td>
</tr>
<tr>
<td></td>
<td>▶ Capable of applying application logic, such as event correlation, which is used to merge or map the event or alarm between the device and enterprise integration layer.</td>
</tr>
<tr>
<td>Devices</td>
<td>Remote sensors or applications, which collect information or perform tasks, that must communicate to and from the central system.</td>
</tr>
</tbody>
</table>
Numerous industry requirements or system constraints might lead a developer to use the client to edge server to enterprise server topology. The main characteristics of systems that can benefit from this topology are listed in Table 4-2.

<table>
<thead>
<tr>
<th>Decision criteria</th>
<th>Client to edge server to enterprise server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use case and topology related</strong></td>
<td></td>
</tr>
<tr>
<td>Data processing at remote location</td>
<td>Moderate to extended (for example, filtering or data exchange between devices at one physical location) Remote locations with data exchange, preprocessing, and filtering require additional mediation capabilities that lead to a dedicated integration endpoint: edge gateway. Examples are retail stores or buildings with smart meter devices.</td>
</tr>
<tr>
<td>Number of devices per location</td>
<td>Multiple and different devices Use an edge gateway to reduce the number of connections and make the data preprocessing more transparent at a single integration endpoint.</td>
</tr>
<tr>
<td><strong>Device related</strong></td>
<td></td>
</tr>
<tr>
<td>Available resources (for example, processor or memory)</td>
<td>None to less resources available on the devices for extensions At remote locations, many devices have limited resources and there is no room for extended integration functionality on these devices. An edge gateway can implement these extensions and reduce the impact on the devices.</td>
</tr>
<tr>
<td>Extensibility (for example, API or connectivity options)</td>
<td>Almost closed device implementations with nearly zero access to functions and data; proprietary device interfaces Often the number of device types is limited to proprietary protocols and connectivity options; a mediation is required in between the devices and the enterprise layer. These mediations can be implemented in a configurable and flexible manner on an edge gateway with extended capabilities for remote monitoring and remote configurability and extensibility.</td>
</tr>
<tr>
<td><strong>Network related</strong></td>
<td></td>
</tr>
<tr>
<td>Reliability of connection</td>
<td>Unreliable connectivity with a moderate to large amount of data that needs to be transferred Unsecured and unreliable networks require a data buffering at remote locations for further delivery. This can be done in an effective manner by an edge gateway. All data buffering and preprocessing can be provided in a transactional context so that no data will be lost or compromised.</td>
</tr>
<tr>
<td>Security of lower protocols</td>
<td>Unsecured connectivity with the need for an additional security layer on top If the device connectivity options are limited to unsecured protocols (for example, FTP or HTTP), the connectivity between remote locations and the enterprise layer is not secured, and an additional security layer is needed on top. This needs to be implemented a single time, at the edge gateway.</td>
</tr>
</tbody>
</table>
4.1.1 The architecture

The solution approach is based on the extension of the Federated Enterprise Service Bus (ESB) Pattern. Figure 4-2 shows a Federated ESB that consists of a backbone service bus and managed integration endpoints, called *edge gateways*. These endpoints help to extend the reach of an ESB to the edge of an enterprise’s IT infrastructure in a reliable and controlled manner.

![Figure 4-2  Client to edge server to enterprise server architecture](image)

Today, many enterprises have already established a central enterprise integration facility and monitoring for technical reasons. Nevertheless, data exchange with branch offices is still handled in a batch-oriented manner with file transfer over unsecured and unreliable connections in many cases. Changing these business processes from batch to reliable connections and near real-time processing requires a paradigm shift. The Federated ESB Pattern helps to extend the control of centrally managed integration processes to the edge.

4.2 Messaging patterns

In software architecture, a messaging pattern is a network-oriented architectural pattern that describes how two different parts of a message passing system connect and communicate with each other. The messaging patterns identify common message flows that are used in messaging solutions. There are five commonly used messaging patterns:

- Fan out broadcast
- Fan in per device notification
- Fan out per device notification
- Fan in per device request-reply
- Fan out per device request-reply
4.2.1 Fan out broadcast

In this messaging pattern, a device or an MQTT client publishes messages on a topic string. The same topic string is also subscribed by multiple MQTT clients. After the message is published by the publisher, a broadcast of the publication is sent to all the subscribers of the topic string. Figure 4-3 shows the architecture of a fan out broadcast messaging pattern.

![Fan out broadcast diagram]

Figure 4-3  Architecture of fan out broadcast messaging pattern

Table 4-3 shows the topic used by different clients based on their roles.

<table>
<thead>
<tr>
<th>Application</th>
<th>Topic</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscriber &lt;1...N&gt;</td>
<td>topic1</td>
<td>Subscribe</td>
</tr>
<tr>
<td>Publisher &lt;1&gt;</td>
<td>topic1</td>
<td>Publish</td>
</tr>
</tbody>
</table>

**Use case for fan out broadcast pattern**

The Australian Open required a scoring solution to provide live scores to millions of fans using various devices. During the 2014 event, the team updated their scoring solution to use MessageSight. The client device application subscribed to topics on which the scores are published. The back-end scoring application published scores on a topic string. The Australian open live score application can be accessed from the following URL.

http://www.ausopen.com/en_AU/scores/

Figure 4-4 on page 93 shows the live score application from the Australian open grand slam event.
4.2.2 Fan in per device notification

In this messaging pattern, many devices having an MQTT client publish messages on a topic string. The same topic string is also subscribed to by a single MQTT client. After different messages are published by the publishers, the published message is routed to the subscriber of the topic string. Figure 4-5 shows the architecture of fan in broadcast messaging pattern.

Table 4-4 on page 94 shows the topic used by different clients based on their roles.
Table 4-4  Application role and enumeration for a fan in per device notification pattern

<table>
<thead>
<tr>
<th>Application</th>
<th>Topic</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscriber &lt;1&gt;</td>
<td>topic1</td>
<td>Subscribe</td>
</tr>
<tr>
<td>Publisher &lt;1...N&gt;</td>
<td>topic1</td>
<td>Publish</td>
</tr>
</tbody>
</table>

**Use case for fan in per device notification pattern**

A waste management company wants to use smart trash cans to notify the trucks to pick up, when the trash cans are getting full. This will help the trash pickup trucks to avoid unnecessary trips for emptying the trash cans. This company can implement the fan in per device notification pattern for smart trash cans to notify the trucks to schedule a trash pickup. Figure 4-6 shows the implementation of this pattern.

![Smart trash cleanup for a waste management company](image)

In this scenario, the trash cans can publish pick up messages on the topic `pickuptrash`. The pickup trucks have subscribed on the same topic and get notified when the trash cans are getting full and they schedule a service pickup for this trash can.

### 4.2.3 Fan out per device notification

In this messaging pattern, a device or an MQTT client publishes messages on multiple unique topic strings; each topic string is also subscribed to by a single MQTT client. This allows a single publisher to send messages to various unique subscribers using unique topic strings. Figure 4-7 on page 95 shows the architecture of fan out per device notification pattern.
Table 4-5 shows the topic used by different clients based on their role.

Table 4-5  Application role and enumeration for a fan out per device notification pattern

<table>
<thead>
<tr>
<th>Application</th>
<th>Action</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscriber 1</td>
<td>Subscribe</td>
<td>topic1</td>
</tr>
<tr>
<td>Subscriber 2</td>
<td>Subscribe</td>
<td>topic2</td>
</tr>
<tr>
<td>Subscriber 3</td>
<td>Subscribe</td>
<td>topic3</td>
</tr>
<tr>
<td>Subscriber 4</td>
<td>Subscribe</td>
<td>topic4</td>
</tr>
<tr>
<td>Subscriber 5</td>
<td>Subscribe</td>
<td>topic5</td>
</tr>
<tr>
<td>Publisher 1</td>
<td>Publish</td>
<td>topic1, topic2, topic3, topic4, topicN</td>
</tr>
</tbody>
</table>

Use case for fan out per device notification pattern

An airline company wants to send notifications to its passengers through the airline mobile application on their smartphones. Each passenger is registered through a frequent flier program and has a unique frequent flier ID. The airline wants to notify a passenger when the flight is ready for web check-in and about their boarding status and their baggage details. Figure 4-8 on page 96 shows the implementation of this pattern for sending push notifications to passengers.
Figure 4-8  Push notifications to passengers for check-in, boarding, and baggage

This implementation shows five passengers who have an airline application installed on their mobile phones. The application on the mobile phones subscribes to a unique notification topic. Each notification topic is unique, because it contains the frequent flier ID of the passenger and it follows the topic pattern as shown in Example 4-1.

Example 4-1  Subscription topic string

FF/<frequent flier ID>/Notify

Table 4-6 shows the various topics on which the applications on the passengers’ mobile phones have subscribed.

Table 4-6  Topic subscribed by passenger application

<table>
<thead>
<tr>
<th>Passenger application</th>
<th>Topic subscribed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passenger 1</td>
<td>FF/1011101/Notify</td>
</tr>
<tr>
<td>Passenger 2</td>
<td>FF/1021101/Notify</td>
</tr>
<tr>
<td>Passenger 3</td>
<td>FF/1093101/Notify</td>
</tr>
<tr>
<td>Passenger 4</td>
<td>FF/1011321/Notify</td>
</tr>
<tr>
<td>Passenger 5</td>
<td>FF/1099321/Notify</td>
</tr>
</tbody>
</table>

If the airline back-end application has to notify passenger 3 for the web check-in, a message must be published on the topic FF/1093101/Notify.

If the airline back-end application has to notify passenger 4 for the baggage claim, a message must be published on the topic FF/1011321/Notify.

By publishing messages on unique topics, the airline company can notify their targeted passengers.
4.2.4 Fan in per device request reply

In this messaging pattern, many devices that have an MQTT client publish messages on different topic strings. The same topic string is also subscribed to by a single MQTT client. The subscriber device publishes reply messages on separate topic strings for each publisher device. The topic strings for request and reply are unique for a combination of different publishers and a common subscriber. Figure 4-9 shows the architecture of the fan in per device request reply messaging pattern.

Table 4-7 shows the topic used by different clients based on their roles.

<table>
<thead>
<tr>
<th>Application</th>
<th>Publisher action</th>
<th>Topic</th>
<th>Subscriber action</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publisher1</td>
<td>Publish request</td>
<td>topic1/request</td>
<td>Subscribe request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topic1/reply</td>
<td>Publish request</td>
<td></td>
</tr>
<tr>
<td>Publisher2</td>
<td>Publish request</td>
<td>topic2/request</td>
<td>Subscribe request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topic2/reply</td>
<td>Publish request</td>
<td></td>
</tr>
<tr>
<td>Publisher3</td>
<td>Publish request</td>
<td>topic3/request</td>
<td>Subscribe request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topic3/reply</td>
<td>Publish request</td>
<td></td>
</tr>
<tr>
<td>Publisher4</td>
<td>Publish request</td>
<td>topic4/request</td>
<td>Subscribe request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topic4/reply</td>
<td>Publish request</td>
<td></td>
</tr>
<tr>
<td>PublisherN</td>
<td>Publish request</td>
<td>topicN/request</td>
<td>Subscribe request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topicN/reply</td>
<td>Publish request</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-7 shows the topic strings for request and reply. The topic strings are unique for a combination of different publishers and subscribers.
Use case for a fan in per device request reply pattern

An airline company wants to allow its passengers using the airline mobile application on their smartphones to query the itinerary and baggage details from their mobile application. Each passenger is registered through a frequent flier program and has a unique frequent flier ID. Figure 4-10 shows the implementation of this pattern for request reply messaging between the passenger mobile application and the airline back-end application.

![Figure 4-10 Request reply messaging pattern between passengers' mobile and back-end applications](image)

This implementation shows five passengers who have an airline application installed on their mobile phones. The application on the mobile phones subscribes to a unique reply topic and publishes on a unique request topic. Each request and reply topic is unique because it contains the frequent flier ID of the passenger and it follows the topic patterns shown in Example 4-2 and Example 4-3.

**Example 4-2 Request topic pattern**

\[ FF/<\text{frequent flier ID}>/\text{QueryRequest} \]

**Example 4-3 Reply topic pattern**

\[ FF/<\text{frequent flier ID}>/\text{QueryResponse} \]

Table 4-8 shows the various topics on which the applications on passengers' mobile phones can publish and subscribe messages.

<table>
<thead>
<tr>
<th>Passenger application</th>
<th>Action</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passenger1</td>
<td>Publish</td>
<td>FF/1011101/QueryRequest</td>
</tr>
<tr>
<td></td>
<td>Subscribe</td>
<td>FF/1011101/QueryResponse</td>
</tr>
<tr>
<td>Passenger2</td>
<td>Publish</td>
<td>FF/1021101/QueryRequest</td>
</tr>
<tr>
<td></td>
<td>Subscribe</td>
<td>FF/1021101/QueryResponse</td>
</tr>
</tbody>
</table>
Table 4-9 shows the various topics on which the airline back-end application will publish and subscribe.

<table>
<thead>
<tr>
<th>Passenger application</th>
<th>Action</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passenger3</td>
<td>Publish</td>
<td>FF/1093101/QueryRequest</td>
</tr>
<tr>
<td></td>
<td>Subscribe</td>
<td>FF/1093101/QueryResponse</td>
</tr>
<tr>
<td>Passenger4</td>
<td>Publish</td>
<td>FF/1011321/QueryRequest</td>
</tr>
<tr>
<td></td>
<td>Subscribe</td>
<td>FF/1011321/QueryResponse</td>
</tr>
<tr>
<td>Passenger5</td>
<td>Publish</td>
<td>FF/1099321/QueryRequest</td>
</tr>
<tr>
<td></td>
<td>Subscribe</td>
<td>FF/1099321/QueryResponse</td>
</tr>
</tbody>
</table>

The Passenger1 application can send a baggage query request on topic FF/1011101/QueryRequest; the message published will be received by the back-end application. The back-end application has already subscribed to receive messages on topic FF/*/QueryRequest. The back-end application will prepare a reply and will publish on topic FF/1011101/QueryResponse. The Passenger1 mobile application has also subscribed on this topic, and, therefore, it will receive the response sent by the back-end application.

4.2.5 Fan out per device request reply

In this messaging pattern, a single device that has an MQTT client publishes messages on a different topic string. Each topic string is also subscribed to by different MQTT clients. The subscriber devices publish reply messages on separate topic strings for the publisher device. The topic strings for request and reply are unique for a combination of a common publisher and different subscribers. Figure 4-11 on page 100 shows the architecture of fan out per device request reply messaging pattern.
Table 4-10 shows the topic used by different clients based on their roles.

Table 4-10  Application role and enumeration for fan out per device request reply pattern

<table>
<thead>
<tr>
<th>Application</th>
<th>Publisher action</th>
<th>Topic</th>
<th>Subscriber action</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publisher</td>
<td>Publish request</td>
<td>topic1/request</td>
<td>Subscribe request</td>
<td>Subscriber1</td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topic1/reply</td>
<td>Publish reply</td>
<td>Subscriber1</td>
</tr>
<tr>
<td></td>
<td>Publish request</td>
<td>topic2/request</td>
<td>Subscribe request</td>
<td>Subscriber2</td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topic2/reply</td>
<td>Publish reply</td>
<td>Subscriber2</td>
</tr>
<tr>
<td></td>
<td>Publish request</td>
<td>topic3/request</td>
<td>Subscribe request</td>
<td>Subscriber3</td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topic3/reply</td>
<td>Publish reply</td>
<td>Subscriber3</td>
</tr>
<tr>
<td></td>
<td>Publish request</td>
<td>topic4/request</td>
<td>Subscribe request</td>
<td>Subscriber4</td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topic4/reply</td>
<td>Publish reply</td>
<td>Subscriber4</td>
</tr>
<tr>
<td></td>
<td>Publish request</td>
<td>topicN/request</td>
<td>Subscribe request</td>
<td>SubscriberN</td>
</tr>
<tr>
<td></td>
<td>Subscribe reply</td>
<td>topicN/reply</td>
<td>Publish reply</td>
<td>SubscriberN</td>
</tr>
</tbody>
</table>

Table 4-10 shows the topic strings for request and reply; the topic strings are unique for a combination of a publisher and different subscribers.

**Use case for a fan out per device request reply pattern**

Smart connected cars are enabled with sensors that can sense a car crash and send requests to emergency services. After the request is received by the emergency services, a reply is sent to notify the connected car that the help is on its way. Figure 4-12 on page 101 shows the implementation of this pattern for request reply messaging between a connected car and a multiple back-end emergency services application.
This implementation shows three emergency services that have subscribed to requests from connected cars:

- Roadside assistance
- Medical and ambulance assistance
- Law enforcement and police assistance

The connected car sends a request to three emergency services and receives individual responses from the emergency services applications. Table 4-11 shows the topics used by the connected car to send a request and receive a reply from the emergency services application.

Table 4-11  Topics for publish and subscribe from connected cars

<table>
<thead>
<tr>
<th>Application</th>
<th>Action</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected car</td>
<td>Publish</td>
<td>sc/CAL001/medicalRequest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sc/CAL001/policeRequest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sc/CAL001/roadsideRequest</td>
</tr>
<tr>
<td></td>
<td>Subscribe</td>
<td>sc/CAL001/medicalReply</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sc/CAL001/policeReply</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sc/CAL001/roadsideReply</td>
</tr>
</tbody>
</table>

Table 4-12 shows the topics used by the emergency services application to receive and reply to a request from a connected car involved at the crash site.

Table 4-12  Topics used by the emergency services application

<table>
<thead>
<tr>
<th>Application</th>
<th>Action</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medical and Ambulance</td>
<td>Subscribe</td>
<td>sc/+/medicalRequest</td>
</tr>
<tr>
<td>Emergency</td>
<td>Publish</td>
<td>sc/CAL001/medicalReply</td>
</tr>
<tr>
<td>Law enforcement and police</td>
<td>Subscribe</td>
<td>sc/+/policeRequest</td>
</tr>
<tr>
<td></td>
<td>Publish</td>
<td>sc/CAL001/policeReply</td>
</tr>
</tbody>
</table>
A smart connected car with registration number CAL001 was involved in a crash. The connected car will immediately send emergency service requests on the following topics:

- \texttt{sc/CAL001/medicalRequest}
- \texttt{sc/CAL001/policeRequest}
- \texttt{sc/CAL001/roadsideRequest}

The emergency services have subscribed to emergency topics from all connected cars using the topic pattern:

- \texttt{sc/+/medicalRequest}
- \texttt{sc/+/policeRequest}
- \texttt{sc/+/roadsideRequest}

The emergency services application will reply to the connected car using the following topics, which are also subscribed to by the connected car application:

- \texttt{sc/CAL001/medicalReply}
- \texttt{sc/CAL001/policeReply}
- \texttt{sc/CAL001/roadsideReply}

Using these topics, the connected car and emergency services application can communicate with each other.

### 4.3 Messaging considerations

In an MQTT-based messaging implementation, consider multiple messaging aspects to develop an efficient system. Several of these considerations are discussed in this section.

#### 4.3.1 Quality of service

MQTT provides three levels of reliability, which are known as qualities of service (QoS). The reliability of the message determines the persistence of the message.

**QoS 0** is delivered at most once:

- QoS 0 is the fastest mode of transfer. It is sometimes called *fire and forget*. The message is delivered at most once, or it is not delivered at all. Its delivery across the network is not acknowledged.
- QoS 0 messages are not persistent. The message is not stored on the IBM MessageSight server. The message might be lost if the publisher is disconnected, or if the server fails. If a subscriber is disconnected at the time the server receives the published message, the subscriber might not receive the message.

**QoS 1** is delivered at least once:

- The message is delivered at least once. Duplicate messages can be delivered.
- If the publisher of the message does not receive an acknowledgment, the publisher resends the message (indicating it is a duplicate) until an acknowledgment is received. As a result, the subscriber can receive the same message multiple times.
QoS 1 messages are persistent. The message is stored on the IBM MessageSight server until the message is sent to all the subscribers of the message. If the message subscriber requested a QoS that requires an acknowledgment, the message is stored until the acknowledgment is received.

QoS 2 is delivered exactly once:
- The message is delivered exactly once. Duplicate messages are not delivered.
- QoS 2 is the safest, but slowest mode of transfer.
- QoS 2 messages are persistent. The message is stored on the IBM MessageSight server until it is sent to all the subscribers of the message. If the message subscriber requested a QoS that requires an acknowledgment, the message is stored until the acknowledgment is received.

In general, higher qualities of service are slower than lower qualities of service.

### 4.3.2 Message size

If the application sends large messages with a high quality of service level, the MQTT server must be configured to store many messages before the available storage is filled. Consider the volume and size of the messages that applications want to send, and whether the messages must be persistent. These decisions affect the number of messages that an MQTT server can handle.

### 4.3.3 Message order

If the message arrival order is important to the application, the solution implementation must ensure that message order is preserved. To preserve the message order, the solution must meet the following requirements:

- The producer or publisher must be single-threaded.
- Messages must be exchanged with the same quality of service.
- If the solution is using MQ Connectivity, it must map to a single queue manager.
- For publish/subscribe systems, the messages must be published to a single topic.

### 4.3.4 Topic namespace

The depth of the topic name space affects the performance of the solution. Therefore, an optimal topic namespace depth should be chosen to avoid performance degradation.

### 4.3.5 Retained message

A publisher application can set retained flag on the published message, this allows MQTT server to send the most recent message to subscribers that subscribe after that message was published. Only one message is retained for each topic, so existing retained messages are deleted when a new retained message is published to the topic. The deletion of the existing message might not occur synchronously with the arrival of the new retained message. In the case where more than one publisher publishes retained messages to a single topic, it is not possible to know which message is stored. It is possible that an earlier message sent by one publisher is stored as the retained message, rather than a later message sent by another publisher. Therefore, where possible, have no more than one publisher sending retained messages to a topic.
IBM MessageSight and the key fob remote application

In this chapter, we present IBM MessageSight security capabilities by describing a practical example using the key fob remote application. This chapter presents the following topics:

- Key fob remote application, including several of the security improvements that can be provided easily by using MessageSight.
- MessageSight appliance is configurable without any coding, and so the implementation of security controls is fast and robust. We describe the authentication and authorization tasks that are performed by the key fob remote application users.
- The security capabilities of the MessageSight appliance on Transport Layer Security (TLS) and how to configure Secure Shell (SSL) or TLS on MessageSight. We explain the configuration of mutual authentication, based on client certificates.

This chapter contains the following topics:

- 5.1, “Overview of the key fob remote application” on page 106
- 5.2, “MessageSight configurations for the key fob remote application” on page 109
- 5.3, “Security capabilities of the MessageSight appliance” on page 119
5.1 Overview of the key fob remote application

This section describes the key fob remote application and describes the MessageSight concepts that are introduced in 3.6, “Message hubs, endpoints, and policies” on page 75.

**Note:** A trial version of the key fob remote application is available at the following URL:

Two online video tutorials related to the key fob application are also available. These describe the MQTT application:

- **IBM MessageSight: Powering an MQTT application**, which is accessible at this website:
  http://bit.ly/keyfobvideo1
- **IBM MessageSight: Authenticating an MQTT application**, which is accessible at this website:

5.1.1 Application overview

The key fob remote application consists of two HTML5 web applications:

- Remote application
- Car application

The Car application models a real vehicle, with audio and text feedback when a control action is made.

The Remote application is a virtual key fob, with the following options:

- Start and stop the car
- Lock and unlock the doors
- Trigger the alarm
- Honk the horn
- Set the internal car temperature

Each application is built using the Eclipse Paho MQTT JavaScript client (see 1.2.4, “The Eclipse Paho project” on page 9). Communication and synchronization occur with the MQTT publish/subscribe messaging. The messaging server used by the MQTT clients to communicate is MessageSight in the following ways:

- The Car application indicates that Bob's car is connected.
- The Remote application indicates that the key fob is connected to Bob's car.
- The communication between the MQTT clients of the two applications (Car and Remote) and the MessageSight server are based on MQTT over websockets.
5.1.2 Testing the key fob remote application

The key fob remote application is available for testing at the following URL:

The first step that is required is to connect both the key fob remote and the Car applications, as shown in Figure 5-2.
Enter the driver name (for example: Bob) in the Driver text field and click **Connect**. You will access these two applications, as shown in Figure 5-3. Follow these steps:

1. The *Remote* application in the left pane.
2. The *Car* application on the right pane.

![Figure 5-3 Remote and Car applications](image)

From here you can become familiar with the key fob, simulated by the *Remote* application, to act on the car, simulated by the *Car* application.

You can perform the following actions on the car:

- Start and stop the car by using the **Start** and **Stop** buttons.
- Lock and unlock the doors by using the **Lock** and **Unlock** buttons.
- Trigger and stop the alarm by using the **Alarm ON/Alarm OFF** buttons.
- Honk the horn by using the **Horn** button.
- Set the internal temperature of the car by using the **Climate Control** selector.

Actions you execute with the *Remote* application display on the *Car* application, as shown in Figure 5-4 on page 109. In this case, the Start engine button has been clicked on the *Remote* application and the text, *Engine started!*, is displayed on the *Car* application.
Chapter 5. IBM MessageSight and the key fob remote application

5.2 MessageSight configurations for the key fob remote application

In this section, we describe the MessageSight configuration that needs to be implemented to make the key fob remote application work.

First, and most importantly, we provide the basic configuration that is required to power a simple MQTT application. Then, we add several security controls to this same configuration.

5.2.1 MessageSight basic configuration

This section describes how to set up and test the basic configuration, which must be implemented to make the key fob remote application work. We presume you have installed a physical or virtual MessageSight appliance, activated the web UI interface on one of the network interfaces of your MessageSight device, and accepted the license agreement.

Set up the basic configuration
From a MessageSight perspective, the configuration has these steps:

1. Connect to the web UI of your MessageSight appliance using a system administrator or messaging administrator account.

2. Select **Messaging ➔ Message Hubs** to access the message hub configuration panel, as shown in Figure 5-5 on page 110.
3. Click the **plus sign** (+) icon to add a new message hub, and a new window opens, as shown in Figure 5-6.

![Figure 5-5 Accessing the Message Hubs configuration page](image)

**Figure 5-5** Accessing the Message Hubs configuration page

4. Enter a name for the message hub (for example, **AutoRemoteHub**) and a description (for example, **Message hub for the key fob remote application**), as shown in Figure 5-7.

![Figure 5-6 Adding a new message hub](image)

**Figure 5-6** Adding a new message hub

5. Click **Save**. The **AutoRemoteHub** message hub is created, but, as shown in Figure 5-8 on page 111, a warning symbol indicates that no endpoint has been configured on the hub.

![Figure 5-7 Setting the message hub name and description](image)

**Figure 5-7** Setting the message hub name and description
6. Select **AutoRemoteHub** (click the pencil icon to edit), as shown in Figure 5-9.

7. At this point, it is necessary to define at least one connection policy and one messaging policy. From the Connection Policies tab of the message hub, click the **plus sign** (+) icon to create a new connection policy, as shown in Figure 5-10.
8. In this scenario, we want to configure a basic message hub, so we define fairly open policies. In the Add Connection Policy window, enter the following configuration parameters, as shown in Figure 5-11, and click **Save**.

![Figure 5-11](image)

The following details of the parameters need to be provided or selected:
- **Name**: Enter a name for the connection, for example, `AutoRemoteConnection`.
- **Protocol**: Select `MQTT`, because we want only MQTT clients to be able to connect to the `AutoRemoteHub` endpoint.
- **Client ID**: Enter `AutoRemote*` because we want to restrict access to the `AutoRemoteHub` endpoint to MQTT clients using a client identifier starting with the `AutoRemote` character string.

9. Next, create a messaging policy. The purpose of a messaging policy is to authorize already connected clients to execute specific messaging actions.

From the Messaging Policies tab of the message hub panel, click the **plus sign** (+) icon to create a new messaging policy, as shown in Figure 5-12.

![Figure 5-12](image)
10. In the opening window, enter the following configuration parameters, as shown in Figure 5-13, and click **Save**.

![Figure 5-13 AutoRemoteMessaging policy configuration parameters](image)

The following details of the parameters need to be provided or selected:
- **Name**: Enter a messaging name, for example, AutoRemoteMessaging.
- **Protocol**: Select MQTT.
- **Client ID**: Enter AutoRemote*.
- **Authority**: Select Publish and Subscribe.
- **Destination**: Enter AutoRemote/*.

The AutoRemoteMessaging messaging policy specifies that connected MQTT clients, with a client identifier starting with the character string, AutoRemote, can both publish and subscribe to any topic in the topic space, AutoRemote/*. A remote publishes commands for a car, using a JavaScript Object Notation (JSON) payload, as shown in Example 5-1.

**Example 5-1 JSON command message to start a connected car**

```
{"action":"START","value":true}
```

These command messages are published on the AutoRemote/<driver_name> control topic, where <driver_name> is the name (in lowercase) of the driver used at connection time, on both the Remote and Car applications.
Each car subscribes to the control topic `AutoRemote/<driver_name>`, receives and processes command messages that are published by a remote, and invokes the appropriate function: start/stop engine, lock/unlock doors, and so on. Then, a car publishes a status update back to the remote key fob. Status updates are published using a JSON payload on the `AutoRemote/<driver_name>/settings` status topic, where `<driver_name>` is the name (in lowercase) of the driver used at connection time, on both the Remote and Car applications.

Each remote subscribes to the status topic `AutoRemote/<driver_name>/settings`, then receives and processes status messages that are published by a car. A remote uses status messages to set the state of its buttons (for example, the Start button is active or inactive) and status display (for example, the Alarm ON/Alarm OFF display on the Alarm button of the remote key fob).

11. The configuration of the AutoRemoteHub message hub finishes with the creation of an endpoint. From the Endpoints tab of the message hub panel, click the **plus sign (+) icon** to create a new messaging policy, as shown in Figure 5-14.

![Figure 5-14 Adding an endpoint on the AutoRemoteHub](image)

12. In the Add Endpoint window, enter the configuration parameters shown in Figure 5-15 on page 115 and described:

- **Name**: Enter an endpoint name, for example, `AutoRemoteEndpoint`.
- **Protocol**: Select **MQTT**.
- **Port**: Enter the value `10000`.  


13. To attach the AutoRemoteConnection connection policy that was configured in Step 8 on page 112, click the plus sign (+) icon in the Connection Policies table of the Add Endpoint panel. Check Add to add the AutoRemoteConnection policy to the AutoRemoteEndpoint endpoint, as shown in Figure 5-16.

14. Click Save to go back to the Add Endpoint panel view.

15. To attach the AutoRemoteMessaging messaging policy configured in Step 10 on page 113, click the plus sign (+) icon in the Messaging Policies table of the Add Endpoint panel (Figure 5-15) and check Add to add the AutoRemoteMessaging policy to the AutoRemoteEndpoint endpoint, as shown in Figure 5-17 on page 116.
Add Messaging Policies to the Endpoint

A messaging policy allows you to control what topics, queues, or global-shared subscriptions a client can access on IBM MessageSight. Each endpoint must have at least one messaging policy. If an endpoint has a messaging policy for a global-shared subscription, it must also have a messaging policy for the subscribed topics.

16. Click **Save** to go back to the Add Endpoint panel view.

17. Confirm that both the **AutoRemoteConnection** and **AutoRemoteMessaging** policies have been added to the **AutoRemoteEndpoint** window, as shown in Figure 5-18, and click **Save**.

18. Click the **Return to Message Hubs** link to return to the list of available Message Hubs on your MessageSight appliance, as shown in Figure 5-19 on page 117.
19. Configuration of the AutoRemoteHub message hub is complete. The warning symbol indicating that no endpoint is attached to the hub is no longer present, as shown in Figure 5-20.

**Test the basic configuration**

To test your MessageSight configuration using the key fob remote application, complete the following steps:

1. Access the key fob remote application using the following URL:
   

2. Enter the connection parameters for both the Remote application (on the left side) and the Car application (on the right side), as shown in Figure 5-21 on page 118, and click **Connect** to connect each application.
The following details describe the most valuable connection parameters for the two applications:

- **Driver name**: Enter a driver name, for example, Bob. Use the same driver's name for the two applications (Remote and Car).
- **Server**: Enter the IP address or host name of your MessageSight appliance.
- **Port**: Enter the listening port of the AutoRemoteEndpoint endpoint, which was created in Step 12 on page 114.

**Note**: The available actions on the Remote application are affected by the status of the Car application. For instance, if the car (or Car application) is disconnected, the remote (or Remote application) cannot operate. As soon as the car is connected to the MessageSight server, the remote is able to connect to the car and send specific control commands.

If you connect the Remote application before connecting the Car application, a message on the Remote application window indicates that the remote cannot connect to the driver’s car, as shown in Figure 5-22 on page 119.
Chapter 5. IBM MessageSight and the key fob remote application

Figure 5-22  Message that the remote cannot connect to the car

As soon as the car is connected, the remote is able to connect to the car, as shown in Figure 5-23.

Figure 5-23  Message that the key fob remote is connected to the car

You can test the key fob remote application with your own MessageSight appliance, as described in 5.1.2, “Testing the key fob remote application” on page 107.

5.3 Security capabilities of the MessageSight appliance

From a MessageSight perspective, the AutoRemoteHub message hub, as defined in 5.2, “MessageSight configurations for the key fob remote application” on page 109, does not provide any security control.
5.3.1 Adding security controls to the key fob remote application

In this section, we discuss certain security controls that can be added for authentication and authorization. We authenticate the clients at connection time and control the actions they can execute when they are connected.

About security in MessageSight
There are three main security aspects of the IBM messaging appliance:

- **Authentication**: Authentication is based on the messaging user ID and password validation. Name-and-password authentication uses a challenge and response protocol to ask messaging users for their names and passwords. This login information is then validated against the information that is stored on the IBM messaging appliance, or stored in an external Lightweight Directory Access Protocol (LDAP) server. User certificates can also be used. The level of authentication is determined by the security profile that is associated with an endpoint.

- **Authorization**: A policy-based authorization mechanism enables messaging users and clients to connect to the IBM messaging appliance and use messaging actions.

- **TLS**: The server certificate and associated security profile controls TLS. The certificate verifies that a public key belongs to a specified entity. The security profile object is used to define the TLS configuration, for example, the minimum protocol method that is allowed when clients connect, or the ciphers that are used for encryption.

User authentication and authorization
When a messaging user connects to the IBM messaging appliance, the messaging user ID and password are authenticated. Any messaging user with a valid messaging user ID and password that is stored on the IBM messaging appliance or on an external LDAP server is successfully authenticated.

If you use authentication to secure your solution, consider complementing it with the appropriate TLS. Using the correct TLS ensures that the user ID and passwords are not transmitted in plain text.

Authentication is not the same as authorization. The authorization mechanism in the IBM messaging appliance is achieved by using policies. These policies enable the messaging user to connect to the IBM messaging appliance and use a messaging action, such as publish or subscribe. Authentication validates that the messaging user is who they claim to be, but it does not allow messaging users to connect or use any messaging actions.

The system administrator can create a messaging user account using the command-line interface (CLI) or the MessageSight web UI.

Messaging users and groups
System administrators can create, edit, and delete IBM messaging appliance users and groups. Alternatively, system administrators can configure an external LDAP server connection to provide the messaging users and groups with information. Messaging users and groups can be authorized by system and messaging administrators to publish and subscribe to topics. We suggest an external LDAP directory for production installations.

Messaging users send and receive messages, but they cannot configure or maintain the IBM messaging appliance.
Each messaging user has a unique client ID. This client ID can be used in a connection policy to allow or deny the user a connection to the IBM messaging appliance. The client ID can be used in a messaging policy to control whether the user can publish or subscribe to a topic and to control whether the user can send, receive, or browse a queue.

Messaging users can be added to groups. Adding users to a group can simplify administration tasks. For example, you might want one set of users to publish to a particular topic, and another set of users to subscribe to a particular topic. Instead of creating individual messaging policies for each user, you can create a messaging group that is authorized to publish to a topic. You can also create another messaging group that is authorized to subscribe to a topic. Users can be added to the relevant messaging group. That group can then be associated with the correct messaging policy.

Managing user authentication and authorization for the key fob remote application

In this section, we configure MessageSight users and groups. Then, we update the AutoRemoteHub endpoint, connection, and messaging policies to allow only the authenticated users to connect and execute messaging actions on their own cars.

Our goal is to permit only those messaging users who belong to the new group that we create to connect to the AutoRemoteHub endpoint.

Create a messaging group and messaging users

To create a messaging group and add messaging users, follow these steps:

1. Connect to the web UI of your MessageSight appliance using a system administrator or messaging administrator account.

2. Select **Messaging** ➔ **Users and Groups** to access the messaging users and group configuration page, as shown in Figure 5-24.

3. Create a single messaging group by clicking the plus sign (+) icon in the Messaging Group section of the page, as shown in Figure 5-25 on page 122.
4. In the opening window, enter the parameters, as shown in Figure 5-26, and click **Save**.

**Figure 5-26   Adding the messaging group named AutoRemoteGroup**

Details about the Group ID parameter follow:
- **Group ID**: Enter a name for the messaging group, for example, *AutoRemoteGroup*. If you choose a different group ID, do not forget to document it, because you will use that name when you modify the configuration of the *AutoRemoteHub* policies and endpoint.

Notice that the new *AutoRemoteGroup* is added to the list of available messaging groups, as shown in Figure 5-27.

**Figure 5-27   AutoRemoteGroup added to the list of messaging groups**

5. Define the messaging users and attach them to the *AutoRemoteGroup*. To create a new messaging user, click the **plus sign** (+) icon in the Messaging Users section of the page, as shown in Figure 5-28 on page 123.
6. In the Add User window, enter the configuration parameters, as shown in Figure 5-29.

- **User ID**: Enter a name of the user, for example, Bob.
- **Password**: Enter a password, for example, Bob, that will be used by messaging user Bob to connect to the AutoRemoteHub endpoint.
- **Confirm Password**: Confirm the password.
- **Group Membership**: Select AutoRemoteGroup, as shown in Figure 5-30 on page 124.
7. Based on Step 5 on page 122, create a second messaging user with the following configuration parameters:
   - **User ID**: The name of a user, for example, sally.
   - **Password**: Enter a password, for example, sally, that will be used by messaging user Sally to connect to the AutoRemoteHub endpoint.
   - **Confirm Password**: Confirm the password.
   - **Group Membership**: Select **AutoRemoteGroup**.

8. Based on step 5 on page 122, create a third messaging user with the following configuration parameters:
   - **User ID**: The name of a user. Enter a name, for example, other.
   - **Password**: Enter a password. This password is used by Messaging User bob to connect to the AutoRemoteHub endpoint. You can choose a password, for example, other.
   - **Confirm Password**: Confirm the password you have chosen, for example, other.
   - **Group Membership**: Do not select any group for this third messaging user.

9. The three messaging users have been added to the messaging user list, as shown in Figure 5-31.

![Figure 5-30 Select a group membership](image)

![Figure 5-31 List of three messaging users](image)
Note: The messaging user other has been created, but it does not belong to a messaging group at this time.

**Update the AutoRemoteHub endpoint, connection, and messaging policies**

Now, we can modify the connection and messaging policies that were defined in Step 13 on page 115 and Step 15 on page 115. We need to provide and manage the user authentication for each user's messaging actions.

To allow user authentication on the AutoRemoteHub message hub, execute the following steps:

1. Connect to the web UI of your MessageSight appliance using a system administrator or messaging administrator account.
2. Select **Messaging → Message Hubs** to access the message hub configuration panel, as shown in Figure 5-5 on page 110.
3. Select the **AutoRemoteHub** message hub, and click the pencil icon to edit this hub, as shown in Figure 5-32.

![Figure 5-32 Edit the AutoRemoteHub message hub](image)

4. On the Connection Policies tab, select the **AutoRemoteConnection** connection policy, and click the pencil icon to edit the policy, as shown in Figure 5-33.

![Figure 5-33 Edit the AutoRemoteConnection connection policy](image)

5. Because we want to restrict the connection to only the users of the AutoRemoteGroup group, check the **Group ID** property, and enter the name of the messaging group **AutoRemoteGroup**, as shown in Figure 5-34 on page 126.
As described in Step 10 on page 113, remotes (or Remote applications) publish on the control topic `AutoRemote/<driver_name>`, which cars (Car applications) subscribe to. Cars publish responses back on the settings topic `AutoRemote/<driver_name>/settings`, which the remotes receive. The name of the driver (`<driver_name>`) is used to connect a remote and a car to the endpoint of the `AutoRemoteHub` message hub.

Setting a messaging policy for every user (in the case of this scenario, the driver) is not possible. Fortunately, MessageSight supports the use of a few substitution variables at policy creation, as described in 3.6.4, "Messaging policies" on page 77.

To substitute the user name used at connection time on the two topics `AutoRemote/<driver_name>` and `AutoRemote/<driver_name>/settings`, we need to create two messaging policies:

- The first policy will be dedicated to the control topic.
- The second policy will be dedicated to the settings topic.

**Enforce messaging policies**

Then, we can use the substitution variable mechanism to enforce the two messaging policies for all of the connected users. To create these two messaging policies and associate them to the `AutoRemoteEndpoint` endpoint of the `AutoRemoteHub` message hub, execute the following steps:

1. Create the messaging policy dedicated to the control topic. From the Messaging Policies tab, click the plus sign (+) icon to create a new messaging policy, as shown in Figure 5-12 on page 112.

2. Enter the configuration parameters, as shown in Figure 5-35 on page 127, and click **Save**.
The following details of the parameters need to be provided:

- **Name**: Enter a name, for example, AutoRemoteControl.
- **Protocol**: Select MQTT.
- **Client ID**: Enter AutoRemote*.
- **Group ID**: Enter AutoRemoteGroup, because we want to apply this messaging policy to connected users.
- **Authority**: Select Publish and Subscribe.
- **Destination**: Select AutoRemote/${UserID}, because we want the control topic to integrate the user name of the connected user.

The AutoRemoteControl messaging policy specifies that connected MQTT clients, with a client identifier starting with the character string AutoRemote, and belonging to the user group AutoRemoteGroup, can both publish and subscribe to the user-specific control topic, AutoRemote/${UserID}.

3. From the Messaging Policies tab, click the plus sign (+) icon to create a new messaging policy, as shown in Figure 5-12 on page 112.

4. Enter the configuration parameters, as shown in Figure 5-36 on page 128, and click Save.
The following details of the parameters need to be provided:

- **Name**: Enter a name, for example, `AutoRemoteSettings`.
- **Protocol**: Select **MQTT**.
- **Client ID**: Enter `AutoRemote*`.
- **Group ID**: Enter `AutoRemoteGroup`, because we want to apply this messaging policy on connected users.
- **Authority**: Select **Publish** and **Subscribe**.
- **Destination**: Select `AutoRemote/${UserID}/settings`, because we want the settings topic to integrate the user name of the connected user.

The `AutoRemoteSettings` messaging policy specifies that connected MQTT clients, with a client identifier starting with the character string `AutoRemote`, and belonging to the user group `AutoRemoteGroup`, can both publish and subscribe to the user-specific settings topic, `AutoRemote/${UserID}/settings`.

5. Check that the two messaging policies have been created, as shown in Figure 5-37 on page 129.
6. Link the control and settings messaging policies to the endpoint of the AutoRemoteHub message hub. From the Endpoints tab, select the AutoRemoteEndpoint endpoint and click the pencil icon to edit the endpoint configuration, as shown in Figure 5-38.

7. In the Messaging Policies section of the Edit Endpoint window, click the plus sign (+) icon to add the control and settings messaging policies, as shown in Figure 5-39.

8. Select the two messaging policies that must be added, AutoRemoteControl and AutoRemoteSettings, as shown in Figure 5-40 on page 130, and click Save.
9. To remove the AutoRemoteMessaging messaging policy from the list of messaging policies that must be enforced on the endpoint, select the **AutoRemoteMessaging** policy and click the **red cross** icon, as shown in Figure 5-41.

10. The final **AutoRemoteEndpoint** endpoint configuration is shown in Figure 5-42 on page 131. Verify the presence of the control and settings policies and click **Save**.
Test the security of your MessageSight configuration

To test the MessageSight configuration and provide a higher level of security, use the key fob remote application to complete the following steps:

1. Access the key fob remote application using the following URL:
   

2. Test the configuration. For example, the user other was created in Step 8 on page 124. Attempt to connect a remote with the user other, who was not given access to AutoRemoteGroup. Confirm that a connection cannot be made.

3. Connect the car first, using an authenticated user, for example, bob, as shown in Figure 5-43, and click Connect to connect the car.

4. The car is connected, as shown in Figure 5-44 on page 132.
5. Connect the remote (Remote application) on Bob’s car with the user other, as shown in Figure 5-45.

6. The user other cannot connect to the AutoRemoteHub message hub because that user does not belong to the AutoRemoteGroup messaging group defined in the connection policy AutoRemoteConnection of the hub’s endpoint, as shown in Figure 5-46 on page 133.
7. Bob’s car is always connected. Now, we perform the same test, but with user sally, as shown in Figure 5-47.

8. This time, the remote displays an error message, indicating that user sally cannot connect to Bob’s car. The user sally can connect to the endpoint because she satisfies the connection policy, but not the messaging policies defined at the endpoint level of the AutoRemoteHub, as shown in Figure 5-48 on page 134. Therefore, sally cannot control Bob’s car.
9. Bob's car is always connected. For the last security test, we try to connect the key fob remote with the user bob, as shown in Figure 5-49.

10. This time the test is successful, the user bob is authenticated and his remote is allowed to access his car to control it, as shown in Figure 5-50 on page 135.
11. To ensure that the remote can operate the controls on Bob's car, click **Start** to start the car engine. As shown in Figure 5-51, Bob's car starts and the status of the car is **started**.

5.3.2 Adding security at the transport level using SSL or TLS

TLS is controlled by the security profile that is associated with an endpoint.

A *security profile* defines the security operations that are applied to a message flow. Each security profile has an associated certificate profile that defines the server certificate to use. *Server certificates* protect against impersonation, certifying that a public key belongs to a specified entity.
TLS is configured within a message hub. Message hubs are associated with endpoints. Endpoints control how clients can connect to the IBM messaging appliance. Each endpoint can be associated with a security profile. The security profile determines the level of encryption and the certificate profile that is used for an endpoint.

The anonymous SSL for the server certificate, key, and mutual authentication is supported on MessageSight. You can specify a user name and password in the connect methods of the Java Message Service (JMS), MQTT, and MQTT over websockets protocols to authenticate users against the local user repository or an LDAP server.

**Client certificate authentication**

Client authentication occurs when MessageSight requests a certificate from a client to verify that the client is who it claims to be.

Figure 5-52 on page 137 is a summary of an SSL and TLS handshake when mutual authentication between an MQTT or JMS client and a MessageSight endpoint is configured on a security profile.

---

**Note:** Information about how to configure an endpoint using SSL or TLS is at the IBM Knowledge Center at the following URL:

Figure 5-52  SSL handshake for mutual authentication

Here is an explanation of the steps of the SSL handshake (Figure 5-52). Mutual authentication is performed in this example:

1. The client sends a client `hello` message that lists the cryptographic capabilities of the client (sorted in client preference order), such as the SSL version, the cipher suites supported by the client, and the data compression methods supported by the client.
   An example of a cipher suite is `TLS_DH_RSA_WITH_DES_CBC_SHA`: the key exchange algorithm is `DH_RSA` (asymmetric). The cipher for data transmission during the session is `DES_CBC` (symmetric). The hash algorithm is `SHA`.
   The message also contains a 28-byte random number, known as `RClientNum` in the following explanations.

2. The server responds with a server `hello` message that contains the cryptographic method (cipher suite) and the data compression method selected by the server, the session ID, and another random number (`RServerNum`). The client and the server must support at least one common cryptographic method (cipher suite), or the handshake fails. The server generally chooses the strongest common cipher suite.

3. The server sends its own digital certificate.
4. The server sends a digital certificate request message to the consumer application, because it requires a digital certificate for client authentication. In this message, the server sends a list of the types of digital certificates that are supported and the distinguished names of acceptable certificate authorities.

5. The server sends a server hello done message and waits for a client response.

6. The client checks the server certificate for the following information:
   – Certificate expiration.
   – Server identity verification, which consists of a verification of the certificate’s signature against a signed certificate in a keystore.

   The client must have the digital certificate of the server in its keystore to validate the server identity. However, it is not always possible to have every server certificate with whom you want to communicate. A better solution for the server is to have its certificate signed by a certificate authority (CA). This way, instead of the server sending its certificate to each client, the client can validate a server certificate when presented during SSL communication by verifying the CA certificate used to sign the server certificate.
   – Check the certification revocation list to see whether the certificate is no longer to be trusted.

7. The client sends a digital certificate. If no suitable digital certificate is available, the client sends a no digital certificate alert. This alert is only a warning, but the server application can fail the session if client authentication is mandatory.

8. The server checks the client certificate in the same way that is described in Step 6.

9. The client sends a digital certificate verify message signed with its private key.

10. By verifying the signature of this digital certificate verify message, the server can explicitly verify the ownership of the client digital certificate with the message that the client is authenticated, if and only if the signature verification is successful. Signature verification is performed using the public key of the client, which is included in the client certificate.

11. The client sends a client key exchange message, encrypted with the public key of the server.

   This message contains the pre-master secret, a 46-byte random number (PMasterSecret) used in the generation of the symmetric encryption keys and the message authentication code (MAC) keys.

   **Note:** An additional process to verify the server digital certificate is not necessary. If the server does not have the private key that belongs to the digital certificate, it cannot decrypt the pre-master secret and create the correct keys for the symmetric encryption algorithm, and the handshake fails.

12. Both client and server use a series of cryptographic operations (using RClientNum, RServerNum, and PMasterSecret) to convert the pre-master secret into a master secret key from which all key material that is required for encryption and message authentication is derived.

13. The client sends a change cipher spec message to make the server switch to the newly negotiated cipher suite, referencing the newly created master secret key (shared between the client and the server only). A change cipher spec message indicates that the contents of subsequent SSL data sent will be encrypted.

14. The next message sent by the client (the finished message) is the first message encrypted with this cipher method and the master secret key.
15. The server sends a change cipher spec message to confirm the use of the newly created master secret key.

16. The next message sent by the server (the finished message) is encrypted using the master secret key.

Communication between the client and server is now secured, and encrypted application data can be sent and received by both the client device and the MessageSight server.

**Configuring mutual authentication on MessageSight**

In the following steps, we detail how to configure mutual authentication based on TLS, between MQTT or JMS clients and a MessageSight server. A private key and a certificate are required for the MessageSight appliance: these cryptographic objects represent the identity of a MessageSight endpoint. A client certificate is also required: the client certificate must be signed by a certificate authority.

The security profile that is created in this chapter is not used in any scenario. The configuration steps are provided to give you the details in case you need to implement such a security configuration.

Follow the steps for configuring mutual authentication on MessageSight:

1. Connect to the web UI of your MessageSight appliance using a system administrator account.

2. Select **Appliance → Security Settings** to access the security settings configuration page, as shown in Figure 5-53.

![Figure 5-53 Accessing MessageSight security settings](image)

3. In the Certificate Profiles section of the Security Settings page, click the **plus sign (+)** icon to create a new certificate profile, as shown in Figure 5-54 on page 140.
4. Enter the configuration parameters, as shown in Figure 5-55, and click **Save**.

The following details describe the parameters that need to be provided:

- **Name**: Enter a name, for example, *MyCertificateProfile*.
- **Certificate**: Click **Browse** and choose the public certificate of the certificate profile. This certificate can be a self-signed certificate.
- **Certificate Password**: Enter the certificate password.
- **Private Key**: Click **Browse** and choose the private key of the certificate profile.
- **Key Password**: Enter the private key password.

5. Check that the new certificate profile *MyCertificateProfile* is added to the list of certificate profiles, as shown in Figure 5-56.
6. Now, we can create a security profile that references the `MyCertificateProfile` certificate profile that will include the elements used to authenticate the client from their certificates.

Click the **plus sign (+)** icon in the Security Profiles section at the bottom of the Security Settings page, as shown in Figure 5-57.

![Figure 5-57 Adding a security profile](image)

7. Enter the configuration parameters, as shown in Figure 5-58, and click **Save**.

![Figure 5-58 Adding a security profile](image)
The following details describe the parameters that can be provided or selected:

- **Name:** Enter a name, for example, `MySecurityProfile`.
- **Minimum Protocol Method:** Select one of the following values:
  - SSLv3
  - TLSv1 (TLS1.0)
  - TLSv1.1
  - TLSv1.2
- **Ciphers:** Select one of the following values:
  - **Best:** Select the most secure cipher supported by the server and the client.
  - **Fast:** Select the fastest high security cipher supported by the server and the client.
  - **Medium:** Select the fastest medium or high security cipher supported by the client.
  - **Client Certificate Authentication:** Check the box if you want to enable client authentication based on SSL or TLS. In our case, we check the box because we want to implement mutual authentication on the security profile.

**Note:** At this step, if you do not want to process mutual authentication but 1-way SSL/TLS only, do not check the Client Certificate Authentication box. Then, save your security profile and go directly to Step 11 on page 143 to discover how to associate your security profile to an endpoint.

- **Use Client Cipher’s Ciphers:** Choose whether to allow the client to determine the cipher suite to use when connecting.
- **Use Password Authentication:** Choose whether to require a valid user ID and password when connecting.
- **Certificate Profile:** Select a certificate profile from the list of available certificate profiles. In our example, we select `MyCertificateProfile`.
- **LTPA Profile:** Do not select any value. For more information about the client authentication mechanism, refer to the IBM Knowledge Center website at the following URL:
  

8. Provide the CA certificate of the client devices, by selecting the `MySecurityProfile` security profile and then selecting **Other Actions → Trusted Certificates**, as shown in Figure 5-59.
9. The opening window is the user interface to use for defining a list of trusted certificates. Click **Browse** to select a client certificate from your file system, as shown in Figure 5-60.

![Figure 5-60 Selecting a trusted certificate](image1)

10. Click **Upload** to upload the selected certificate from the list of MessageSight trusted certificates, as shown in Figure 5-61, and click **Close**.

![Figure 5-61 Certificate uploaded in the list of trusted certificate](image2)

**Note:** The certificate added in the truststore must be an X.509 certificate and signed by a certificate authority (CA) trusted by MessageSight. A self-signed certificate is not accepted as a trusted certificate.

11. To associate the **MySecurityProfile** security profile to an endpoint, it is necessary to edit this endpoint. On the Edit Endpoint page, select the security profile to associate to the current endpoint (**MySecurityProfile** in this example), as shown in Figure 5-62 on page 144, and click **Save** to register the new endpoint configuration.
Revoking certificates
To revoke a certificate, you must replace it with a new certificate on the MessageSight appliance. Delete the certificate profile of the compromised certificate, and create a certificate profile with a new certificate. Then, associate the new certificate profile with the correct security policy.

Federal Information Processing Standards (FIPS)
The National Institute for Standards and Technology (NIST) is a government organization that is concerned with IT systems and security. NIST produces recommendations and standards, including the Federal Information Processing Standards (FIPS).

The FIPS 140-2 standard requires the use of strong cryptographic algorithms. FIPS 140-2 also specifies the requirements for hashing algorithms for use in protecting packets against modification in transit. Over time, analysts develop attacks against existing encryption and hashing algorithms. New algorithms are adopted to resist those attacks. FIPS 140-2 is periodically updated to include these changes. For more information about FIPS, see the following website:

http://www.itl.nist.gov/fipspubs/

Note: You can have the IBM messaging appliance configured to provide FIPS 140-2 support. Changes to the FIPS 140-2 setting require a restart of the MessageSight server.

To provide FIPS 140-2 support on a MessageSight appliance, execute the following steps:
1. Connect to the web UI of your MessageSight appliance using a system administrator account.
2. Select Appliance → Security Settings to access the message hub configuration panel, as shown in Figure 5-63 on page 145.
3. In the System-wide Security Settings section of the Security Settings page, check **Use FIPS 140-2 profile for secure messaging communications**. A window opens, indicating that a restart of the MessageSight server is required for the modification to take effect, as shown in Figure 5-64.

4. Click **Set and Restart** to provide FIPS support on your MessageSight appliance.
Overview of the PickMeUp application

This chapter provides an overview of the PickMeUp in the form of a scenario.

We describe the following topics as they relate to the scenario:

- 6.1, “Company A scenario” on page 148
- 6.2, “PickMeUp architecture for real-life use” on page 153
- 6.3, “Company A solution” on page 163
6.1 Company A scenario

Mobile technology has been significantly changing the way organizations service their customers. The increased use of mobile technology is forcing enterprises to innovate, transform, and to try new ways of reaching, engaging, and communicating with customers. Enterprises can benefit by using the unique features offered by mobile platforms. In this chapter, we define a brief scenario and apply the requirements and architecture of the PickMeUp application to determine how beneficial this application can be in this scenario.

6.1.1 Company A business problem

Company A, a taxi company, wants to improve its quality of service and productivity by automating its business processes. They can do this by taking advantage of mobile technologies. Their need is to connect a passenger who needs a taxi with one of the company’s taxi drivers, and to do so in the most convenient and effective way possible. The traditional, centralized call center model is not efficient in many cases, because delays or errors can occur at the coordinator endpoint that can lead to unexpected mishaps between a passenger and driver.

Throughout this chapter, we refer to the business problem of Company A and the solution as provided by the PickMeUp application.

6.1.2 Requirements for the PickMeUp application at Company A

There are two categories of requirements for PickMeUp. In this section, we describe these requirements and apply them to the Company A scenario described in 6.1, “Company A scenario” on page 148:

- Functional requirements
- Non-functional requirements

Functional requirements for Company A

The functional requirements for the Company A scenario (see 6.1, “Company A scenario” on page 148) are listed:

- The passenger’s mobile device (referred to in this book as the passenger app)
- The driver’s mobile device (referred to in this book as the driver app)

Figure 6-1 on page 149 visually captures the functional requirements for PickMeUp for use at Company A.
Actors

There are five main actors in the context of PickMeUp in our scenario for Company A:

- **Driver**: Taxi driver at Company A
- **Passenger**: Customer of Company A
- **IBM MessageSight server**: MessageSight messaging server
- **Payment system**: Payment system that processes payments for taxi rides
- **PickMeUp back end**: Server side of PickMeUp
**Company A driver actor**

The requirements of the driver actor are listed:

- **Login**
  
  The driver logs in to the PickMeUp driver-based application (the driver app), using a registered user name for authentication. Successful authentication connects the driver with the MessageSight server from which to perform a variety of tasks.

  **Note:** In the mobile app, the driver signs in only with a user name (no password is required) to connect to the MessageSight server.

- **List of current PickMeUp requests**
  
  After logging in to the driver app, the driver can list all local, pending PickMeUp requests from passengers. The list is filtered by PickMeUp, based on the driver's current position, so that the driver can only see the requests in a particular area.

  **Note:** The app lists all of the local, pending PickMeUp requests that are currently in the system.

- **Accept a PickMeUp request**
  
  The driver needs the ability to accept a PickMeUp request using the PickMeUp driver application. Only one PickMeUp request can be accepted by the driver at a time.

- **Chat**
  
  The driver can chat with the passenger in real time. In this case, there are four components of PickMeUp chats:
  
  - **Send text:** The driver can enter and send text content to a passenger using the chat feature. In the scope of the demonstration application, PickMeUp supports English, but multi-language support can be considered.
  
  - **Send voice:** The driver can record a voice message (a voice chat) to send to the passenger during a chat.
  
  - **Send picture:** The driver can take a picture from his device running the PickMeUp and send it to the passenger.
  
  - **Play multimedia content:** The driver can receive and play back the voice chat that is sent from passenger.

- **View journey map**
  
  The driver can view a map simulating the current journey. The journey includes two parts:
  
  - **Locating the passenger:** After accepting a PickMeUp request from a passenger, the driver can view a map to locate the position of the passenger. The map can show the estimated distance to the location and the estimated time of arrival.
  
  - **Riding:** After picking up the passenger, the driver can view the ride journey on the map on the driver PickMeUp app. The map shows the travel distance, time, and cost.
**Company A passenger actor**

The requirements of the passenger actor are listed:

- **Login**
  
  To connect to the MessageSight server, the passenger logs in to the PickMeUp passenger app, using the registered user name for authentication.

  **Note:** In the mobile app, the driver signs in only with a user name (no password is required) to connect to the MessageSight server.

- **Submit a PickMeUp request**

  When logged in to PickMeUp, the passenger can submit a PickMeUp request.

- **Chat**

  The passenger is able to chat with the paired driver in a real-time manner. This use case includes four other use cases:

  - **Send text:** The passenger can enter and send text content to the driver with whom the passenger is chatting. In the scope of the demonstration application, we only support English, but in a real use case, multi-language support is being considered.

  - **Send voice:** The passenger can make a voice recording (a voice chat) and send it to the driver during a chat.

  - **Send picture:** The passenger can take a picture from the device that is running PickMeUp and send it to the driver.

  - **Play multimedia content:** This use case illustrates that the driver can play back the voice chat content that was sent from the passenger.

- **Make a payment**

  When the trip is finished, the passenger can initiate a payment using the payment information that was used to register with PickMeUp.

  **Note:** Processing a typical payment is easy, following an initial registration process.

**MessageSight server actor**

The requirements of the MessageSight server actor are listed:

- **Authenticate and authorize**

  The MessageSight server receives a secure MQTT connection request from PickMeUp and performs authentication and authorization processes to ensure that a secure connection is established between the mobile application and the IBM MessageSight server. For more details about authentication and authorization, see “About security in MessageSight” on page 120.

- **List PickMeUp requests**

  The IBM MessageSight server receives publications from the passenger app about specific topics for PickMeUp requests. These messages are published to the driver app, which has been subscribed to matching topic names. This enables the driver to list the PickMeUp requests on the driver's device.
Accept a PickMeUp request

The IBM MessageSight server receives publications from the PickMeUp driver app, on a specific topic associated with the passenger. This message is then published to the PickMeUp passenger app, which has subscribed to a matching topic name, enabling the driver and passenger to start the ride share process.

Chat

This use case can be initiated either by the driver or the passenger. The IBM MessageSight server receives a publication from PickMeUp, on a specific topic associated with either the passenger or the driver. The message is then published to the driver or passenger app, which is subscribed to the matching topic. The message can be text, picture, or voice.

Submit a PickMeUp request

The IBM MessageSight server receives a publication from the PickMeUp passenger app on a specific topic associated with the passenger. The message is then published to the PickMeUp driver app, which is subscribed to the matching topic, and pending requests that originate from other passengers are displayed, as well.

View journey map

The IBM MessageSight server receives publications from PickMeUp mobile applications about a specific topic, for changes to the geographic location of the passenger and the driver. These messages are later published to the PickMeUp driver or passenger app, which is subscribed to the matching topic, to receive the location information for map viewing and updating.

Make payment

After the passenger has arrived at the destination, and the share ride trip has ended, the IBM MessageSight server receives a publication from the PickMeUp passenger app. The message is then published to the back-end component, which is subscribed to the specific topic associated with payment request. Further steps are displayed for the payment process.

Payment system actor

After the Make Payment request is received from the PickMeUp back end, the payment system actor verifies the payment information provided by the passenger, then proceeds with the payment transaction. The payment result is forwarded to the PickMeUp back end for the next steps. See the next section.

PickMeUp back-end actor

After the request to make a payment is received from the passenger, the PickMeUp back end interacts with the Payment system to perform the transaction. The result is communicated to both the driver app and the passenger app, through the MessageSight server, in the form of MQTT messages and publications.

Non-functional requirements for Company A

This section describes the non-functional requirements for PickMeUp, including the service levels the system must satisfy. These requirements apply to the Company A scenario (6.1, “Company A scenario” on page 148) and to other general scenarios, as well.

- Capacity and performance
- Availability
- Security
**Capacity and performance**
The PickMeUp system must have the ability to handle a high number of requests and messages.

Given the need for a near real-time experience, the messages must be transferred and processed fast, with low latency between endpoints.

**Availability**
The system must be available 24 hours/day, seven days/week.

**Security**
Every connection and transaction between endpoints must be secured.

### 6.2 PickMeUp architecture for real-life use

This section describes a general architecture overview of PickMeUp. It is applicable to the Company A scenario (see 6.1, “Company A scenario” on page 148) and to any general scenario for PickMeUp. This section describes these topics:

- PickMeUp architecture overview
- Considerations for real-life use of PickMeUp

#### 6.2.1 Architecture overview

This section describes the overall architecture of PickMeUp in an attempt to propose preferred practices for developing mobile applications with MQTT and MessageSight.

Figure 6-2 on page 154 captures the overall architecture of the PickMeUp system and briefly describes the anticipated software and hardware components that are required to implement the solution in the scope of this book.
The architecture consists of the following components:

- **Passenger app**
  
The PickMeUp passenger app runs on the passenger's mobile device.
  
The app is built to be an MQTT client application. The target platforms for the app, within the scope of this book, are Android, iOS, and mobile web with HTML5 enabled. There is one passenger app for each of the platforms. The iOS version uses the IBM WebSphere MQ Client Pack as the MQTT client library. The Android version uses the Eclipse Paho library. The web version uses the Eclipse Paho JavaScript client library.

**Note:** Native mobile application development has been chosen to take advantage of OS-specific features and, because there are three versions of the app on various platforms, to demonstrate the abundance of the MQTT clients available for developers to use in the solutions.

- **Driver app**
  
The PickMeUp driver app runs on the driver's mobile device. The target platforms for the app are iOS and mobile web with HTML5 enabled.

**Note:** To quickly try the application without any installation, having an HTML5 version of the driver app is critical for running the app directly on mobile browsers that support HTML5.
Chapter 6. Overview of the PickMeUp application

• **PickMeUp back end**
  The PickMeUp back end is designed to act as an interface to seamlessly simulate the payment process. The back end is also responsible for cleaning up the abundant data that is stored on the MessageSight server.

  **Note:** To simplify development, deployment, and hosting, the back end is built using Node.js and the MQTT node library on top of the IBM Bluemix™ cloud environment.

• **IBM MessageSight**
  The MessageSight appliance functions as the messaging server for the solution.

• **MQTT**
  MQTT is used as the messaging protocol over the MessageSight server to provide real-time experience and a high messaging capacity.

### 6.2.2 Considerations for the real-life use of PickMeUp

This section describes several considerations when you use MQTT and MessageSight for developing a real-time mobile application in actual cases:

• Referential architecture
• Security model

**Referential architecture**
This section provides a typical architecture overview to use as a referential architecture for building a real mobile application with MQTT and MessageSight. Figure 6-3 on page 156 is the architecture overview diagram for such a use case.
In Figure 6-3, notice the following areas:

- The back end is more comprehensive for covering the integration with other enterprise applications and subsystems.
- MQ Connectivity can be used to connect MessageSight with the WebSphere MQ network for the usage of MQTT and messaging among enterprise systems.
- Other enterprise resource planning applications can be integrated to make the application work better with other parts of the enterprise, for example, a payment application or an accounting application.
- Enterprise Lightweight Directory Access Protocol (LDAP) can be used to authenticate system users.
- Single sign-on (SSO) support can be achieved by adding Lightweight Third-Party Authentication (LTPA) or an open standard for authorization (OAuth) service provider to the architecture. This will enhance the authentication capabilities of the system.
- MessageSight can be placed at the enterprise side or DMZ zone.

Security considerations

In this section, we deal with improving the security level of the PickMeUp solution.

The MessageSight appliance plays a prominent role in the PickMeUp solution, because it (MessageSight) acts as a hub between the various actors of the solution. Therefore, MessageSight is a key component in terms of security.
A basic PickMeUpHub message hub configuration is described in “Create a basic message hub using the CLI” on page 82.

**Improve the security level of the driver application**

In this section, we present a solution for authenticating the driver app and the drivers on MessageSight at connection time.

The following security improvements can be enhanced for the driver app and for drivers:

- To ensure that a driver who connects to the system is using the official driver app, mutual authentication is carried out. This application can be reserved for drivers of a specific taxi company.

  Mutual authentication can be based on Secure Shell (SSL) (v3) or Transport Layer Security (TLS) (v1.0, v1.1, or v1.2). Both SSL and TLS are supported on MessageSight when mutual authentication needs to be implemented.

  Configuration of mutual authentication on MessageSight is described in “Configuring mutual authentication on MessageSight” on page 139.

- Authentication of drivers that use a user name and password. After the driver app has been authenticated, the driver needs to be authenticated at connection time. The user name and password capabilities of MQTT can be used, because MessageSight can rely on an LDAP server and perform the required security enforcement. The drivers, as messaging users, can also be configured directly on MessageSight and can be associated to user groups.

  **Note:** Configuration details are available in Configuring an external LDAP server in the IBM MessageSight Knowledge Center:

  [Note link](http://www-01.ibm.com/support/knowledgecenter/SCCGQ_1.1.0/com.ibm.ism.doc/Administering/ad00371_.html?lang=en)

A MessageSight endpoint with a dedicated security profile and a connection policy can be configured to support mutual authentication of the driver app and authentication of the drivers.

Another important aspect of security is authorization. The list of driver topics for both publications and subscriptions is in 7.4.1, “Driver app topics” on page 186. Based on these topics defined for the PickMeUp messaging use case, it is possible to configure messaging policies to enforce them on MessageSight, after the drivers are connected. The goal here is to confirm that the drivers use a dedicated list of topics for both publish and subscribe actions.

Driver messaging policies can be configured with the destinations and authorities shown in Table 6-1 on page 158.
Table 6-1  Destination and authority of MessageSight messaging policies for drivers

<table>
<thead>
<tr>
<th>Topic defined at design time</th>
<th>Destination</th>
<th>Authority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickmeup/drivers/driverId</td>
<td>Pickmeup/drivers/${ClientId}</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/requests/passengerId</td>
<td>Pickmeup/requests/*</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/inbox</td>
<td>Pickmeup/passengers/*/*inbox</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/chat</td>
<td>Pickmeup/passengers/*/*chat</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/location</td>
<td>Pickmeup/drivers/${ClientId}/location</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/inbox</td>
<td>Pickmeup/drivers/${ClientId}/inbox</td>
<td>Subscribe</td>
</tr>
<tr>
<td>Pickmeup/requests/+</td>
<td>Pickmeup/requests/*</td>
<td>Subscribe</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId</td>
<td>Pickmeup/passengers/*</td>
<td>Subscribe</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/picture</td>
<td>Pickmeup/passengers/*/*picture</td>
<td>Subscribe</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/location</td>
<td>Pickmeup/passengers/*/*location</td>
<td>Subscribe</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/chat</td>
<td>Pickmeup/drivers/${ClientId}/chat</td>
<td>Subscribe</td>
</tr>
</tbody>
</table>

Note: The destination property uses the substitution variable `${ClientId}`. Details about substitution variables are available in 3.6.4, “Messaging policies” on page 77.

Figure 6-4 summarizes a possible MessageSight endpoint configuration for enhancing the security of the PickMeUp solution for drivers.

Figure 6-4  Driver endpoint properties
**Improve the security level of the passenger app**

In this section, we present a solution for improving the security level of the PickMeUp solution from a passenger perspective.

Passengers do not need to be authenticated when they request a ride. The only stage when passengers need to be authenticated is when payment is being made. Therefore, we need two endpoints that are dedicated to passengers on MessageSight:

- The first endpoint must provide confidentiality between the passenger and driver. Therefore, 1-way SSL or TLS is required at connection time on this endpoint. The same security level can also be used during the pairing, approaching, and riding stages.

- The second endpoint is dedicated to payment. To connect to this endpoint, passengers first authenticate to a web token service and then use the delivered token to authenticate through a configured MessageSight payment endpoint.

To authenticate passengers, MessageSight can rely on LTPA tokens.

Another important aspect of security is authorization. The list of passenger topics for both publications and subscriptions is in 7.4.2, “Passenger app topics” on page 186. Based on these topics, as defined for the PickMeUp messaging use case, it is possible to configure messaging policies to enforce them on MessageSight, after a passenger is connected. The goal is to confirm that a passenger is using a dedicated list of topics for both publish and subscribe actions.

Passenger messaging policies can be configured with the destinations and authorities shown in Table 6-2.

<table>
<thead>
<tr>
<th>Topic defined at design time</th>
<th>Destination</th>
<th>Authority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickmeup/passengers/passengerId</td>
<td>Pickmeup/passengers/${ClientId}</td>
<td>publish</td>
</tr>
<tr>
<td>Pickmeup/requests/passengerId</td>
<td>Pickmeup/requests/${ClientId}</td>
<td>publish</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/picture</td>
<td>Pickmeup/passengers/${ClientId}/picture</td>
<td>publish</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/location</td>
<td>Pickmeup/passengers/passengerId/location</td>
<td>publish</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/chat</td>
<td>Pickmeup/drivers/*/chat</td>
<td>publish</td>
</tr>
<tr>
<td>Pickmeup/payments</td>
<td>Pickmeup/payments</td>
<td>publish</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/inbox</td>
<td>Pickmeup/passengers/${ClientId}/inbox</td>
<td>subscribe</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId</td>
<td>Pickmeup/drivers/*</td>
<td>subscribe</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/picture</td>
<td>Pickmeup/drivers/*/picture</td>
<td>subscribe</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/chat</td>
<td>Pickmeup/passengers/${ClientId}/chat</td>
<td>subscribe</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/location</td>
<td>Pickmeup/drivers/*/location</td>
<td>subscribe</td>
</tr>
</tbody>
</table>

**Note:** Destination properties use the substitution variable ${ClientId}. Details about substitution variables are available in 3.6.4, “Messaging policies” on page 77.

Figure 6-5 on page 160 summarizes a possible MessageSight endpoint configuration for enhancing the security of the PickMeUp solution for passengers at the connecting, pairing, approaching, and riding stages.
Figure 6-5  Passengers driving endpoint properties

Figure 6-6 on page 161 summarizes a possible MessageSight endpoint configuration for enhancing the security of the PickMeUp solution for passengers at the payment stage.

MQTT - Publish
pickmeup/passengers/[$ClientId]
pickmeup/requests/[$ClientId]
pickmeup/passengers/[$ClientId]/picture
pickmeup/passengers/[$ClientId]/location
pickmeup/drivers/*/chat

MQTT - Subscribe
pickmeup/passengers/[$ClientId]/inbox
pickmeup/drivers/*/picture
pickmeup/passengers/[$ClientId]/chat
pickmeup/drivers/*/location
In this section, we present a solution for improving the security level of the PickMeUp solution from a back-end perspective.

We can use mutual authentication to authenticate the back-end application on MessageSight. The configuration of mutual authentication on MessageSight is described in “Configuring mutual authentication on MessageSight” on page 139. After the back-end application is authenticated, it can publish and subscribe on specific topics as defined at design time and presented in 7.4.3, “Back-end application topics” on page 187.

A MessageSight endpoint with a dedicated security profile and messaging policies can be configured to support mutual authentication and authorization of the back-end application.

The back-end messaging policies can be configured with the destinations and authorities shown in Table 6-3.

<table>
<thead>
<tr>
<th>Topic defined at design time</th>
<th>Destination</th>
<th>Authority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickmeup/drivers/driverId/inbox</td>
<td>Pickmeup/drivers/*/inbox</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/inbox</td>
<td>Pickmeup/passengers/*/inbox</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/location</td>
<td>Pickmeup/passengers/*/location</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId</td>
<td>Pickmeup/passengers/*/</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/picture</td>
<td>Pickmeup/passengers/*/picture</td>
<td>Publish</td>
</tr>
<tr>
<td>Pickmeup/payments</td>
<td>Pickmeup/payments</td>
<td>Subscribe</td>
</tr>
</tbody>
</table>

Figure 6-6  Passengers payment endpoint properties

**Improve the security level of the back-end application**

The configuration of mutual authentication on MessageSight is described in “Configuring mutual authentication on MessageSight” on page 139. After the back-end application is authenticated, it can publish and subscribe on specific topics as defined at design time and presented in 7.4.3, “Back-end application topics” on page 187.
Figure 6-7 summarizes a possible MessageSight endpoint configuration for enhancing the security of the PickMeUp solution for the back-end application.

![Figure 6-7](image)

**Overview of the complete solution**

In this section, we provide an overview of the complete PickMeUp secured solution. This solution integrates different security enhancements that have been proposed to secure the driver, passenger, and back-end applications.

Figure 6-8 on page 163 depicts the complete, secured PickMeUp solution.
6.3 Company A solution

This section provides an overview of the solution that was described in 6.1, “Company A scenario” on page 148 and how the solution can be implemented. This solution overview addresses the following needs:

- “Company A business problem” on page 148
- “Functional requirements for Company A” on page 148
- “Non-functional requirements for Company A” on page 152

We address the business problem and the requirements in the following areas:

- Connecting
- Pairing
- Approaching
- Riding
- Processing payment
### 6.3.1 Scenario 1: Connecting

We describe how authentication and authorization can be achieved for MQTT clients that are securely connected with MessageSight server. This scenario uses the PickMeUp MQTT mobile client that is developed specifically for drivers and passengers on the Android and iOS platforms:

1. The passenger logs in to the PickMeUp passenger app, with the user name and password that are registered with the PickMeUp system.
2. PickMeUp uses the entered credentials to submit a request for an LTPA token. This request is sent to the PickMeUp web component that runs on IBM WebSphere Application Server for authentication.
3. WebSphere Application Server validates the user name and password against the enterprise LDAP server, authenticates the driver, and provides the LTPA token back to the PickMeUp mobile application.
4. PickMeUp uses the token value as the password in the connection object of the MQTT client configuration in the application and then requests that a connection with the MessageSight server be established.
5. The MessageSight server decrypts the LTPA token and retrieves the user name and token expiration time. MessageSight then validates the user and retrieves the required information, such as group information, to confirm which actions the user is authorized to perform.
6. PickMeUp authenticates and authorizes both the driver app and the passenger app. The connection from the application to MessageSight is established.

For simplification, and so that readers can quickly build and run the code, the PickMeUp mobile application developed within the scope of this book does not require the level of security described here for connecting to the MessageSight server. With that consideration, the passenger and driver only need to enter their user name for the app to be connected with the server.

Figure 6-9 shows the UI on the passenger and driver apps during this connection stage.

![Figure 6-9  Passenger app and driver app UI during the connecting stage](image-url)
6.3.2 Scenario 2: Pairing

Scenario 2 describes the pairing process between the PickMeUp driver app and passenger app:

1. The passenger submits a PickMeUp request using the passenger app. The request is transformed to an MQTT message, which includes the current location of the passenger and the publication to a specific topic name on the MessageSight server.

2. The driver receives the PickMeUp request using the driver app, which subscribes to the matching MQTT topic name on the MessageSight server. The request to the driver is in the form of an MQTT message. When delivered to the driver app, the message displays in a list of requests and on a map of the passenger's physical location.

3. The driver accepts a request in the list. The acceptance is transformed to an MQTT message that includes information about the passenger and publication to the specific topic name on the MessageSight server on which the passenger app subscribes.

4. The passenger is notified that a driver has been located and the passenger is asked to continue.

5. The driver app and the passenger app are both switched to the next stage.

Figure 6-10 shows the UI of the passenger app and the driver app in the pairing scenario.

![Figure 6-10 Passenger app and driver app UI during the pairing stage (© OpenStreetMap contributors)](image)

6.3.3 Scenario 3: Approaching

This scenario uses the PickMeUp mobile application for the driver and passenger to locate each other using the MessageSight server:

1. The driver app sends the information about the driver to the MessageSight server, including the current location, a picture, and chat content. The information is in the form of MQTT messages and publications on a specific set of topic names that are associated with the passenger and on which the passenger app is subscribing. An estimated distance from the driver's location to the passenger's location displays, with an estimated arrival time.
2. The passenger app sends information about the passenger to the MessageSight server, including the current location of the passenger and chat content. Chat content can include text, voice, and an optional photograph from the passenger's profile. The information is in the form of MQTT messages and publications on a specific set of topic names that are associated with the driver and on which the driver app is subscribing. The passenger can switch between chat view and map view on the passenger app.

3. The MessageSight server delivers the messages to and from the driver app to the passenger app in real time. Messages are updated in the UI to include the changing position of the driver. If the passenger moves to a new location, that position is updated, as well.

4. As the driver approaches the passenger, both the passenger app and driver app are switched to the next stage.

Figure 6-11 shows the UI of the passenger app during the approaching stage.
6.3.4 Scenario 4: Riding

For the riding stage, PickMeUp is used for the driver and passenger to visualize the journey to the destination.

1. After the ride has started, the driver app is switched to riding mode with a full-screen map. In this mode, the driver app updates the current location on the map, with distance traveled, time traveled, and cost so that the driver and the passenger can both see the trip visualized in real time.

2. The passenger app stays in riding mode during the trip, waiting for a signal from the MessageSight server to switch to the next mode.

3. After their arrival at the destination, the driver ends the trip using the driver app. The trip-end event is then transformed to an MQTT message and publication on the specific topic name to trigger the next stage.

4. Both the passenger app and driver app receive the trip-end signal in the form of an MQTT message and publication through MessageSight and switch to the payment stage.

Figure 6-13 on page 168 shows the UI of the driver app in the riding stage.
6.3.5 Scenario 5: Payment

This scenario uses the PickMeUp driver app, passenger app, and a simulated PickMeUp back end to describe the payment process.

1. With the ride over, the driver requests a payment using the driver app. The request is in the form of an MQTT message and publication on the specific topic name on the MessageSight server. Simultaneously, information about the trip is displayed on the driver app, which is waiting for a signal to switch to next stage.

2. MessageSight delivers the payment request to the passenger app and awaits confirmation to continue.

3. The passenger agrees to make the payment with additional information, such as tip and rating. The confirmation and additional payment information are transformed to an MQTT message and publication on a specific topic name on the MessageSight server on which the PickMeUp back end subscribes.
4. The PickMeUp back end receives the message, parses the payment details, and then sends notification to the passenger app and driver app indicating that payment is completed. The notification is in a form of an MQTT message and publication to specific topic names that are associated with the passenger and driver on which the passenger app and driver app are subscribing.

Note: In an actual scenario, this step might differ. For example, the PickMeUp back end receives the message, parses the payment details, and then interacts with other payment subsystems to perform the payment. After the payment transaction is complete, the back end sends a notification to the passenger app and driver app that payment is completed. The notification is in the form of an MQTT message and publication to specific topic names that are associated with the passenger and driver on which the passenger app and driver app are subscribing.

5. The passenger app and driver app receive a notification that the transaction is complete. The notification originates from the PickMeUp back end, which interacts with the Payment system. All messages and notifications are transferred to endpoints by MessageSight. Figure 6-14 shows the UI of the driver app and the passenger app in the payment scenario.

With the transaction complete, the PickMeUp back end performs a cleanup of MessageSight against the completed transaction to ensure that no extraneous data is left on MessageSight. The passenger app and driver app display the main UI, from which a new PickMeUp cycle can be started.
PickMeUp messaging scenario

This chapter discusses the PickMeUp solution design from a messaging standpoint.

The transport protocol that is implemented on the PickMeUp solution is MQTT, an extremely lightweight publish/subscribe messaging protocol.

The MQTT topics and messages that are implemented in the solution are detailed in this chapter. We present the stages of the PickMeUp scenario, the actors involved, and the topics and messages used for both publications and subscriptions.

Other considerations, such as quality of service (QoS) level, session type for each subscription (durable or non-durable), and publication type (retained or not retained), are also described in this chapter.

The chapter includes the following topics:

- 7.1, “Actors in the PickMeUp solution” on page 172
- 7.2, “Stages of the PickMeUp solution” on page 173
- 7.3, “Topics and messages for the PickMeUp scenario” on page 174
- 7.4, “Summary of publication and subscription topics for the PickMeUp solution” on page 185
7.1 Actors in the PickMeUp solution

In this section, we present the actors of the PickMeUp solution. A detailed description of the actors, as applied to a scenario, is included in “Actors” on page 149.

There are four main actors in the PickMeUp solution:

- **MQTT broker or hub**: This component is the keystone of the solution. All of the messages that are exchanged transit through this hub. This component is implemented using an IBM MessageSight appliance.
  
  The MQTT broker needs to provide all of the security mechanisms to authenticate clients at connection time, and to manage authorizations after the clients are connected. The broker also needs to provide security levels on the transport layer.

- **Drivers**: Drivers connect to the MQTT broker using a specific driver application (referred to as a driver app in this chapter) that embeds an MQTT client. The driver app can publish/subscribe to specific topics on the MQTT broker. Drivers do not connect directly with other actors, other than the messaging server (IBM MessageSight).

- **Passengers**: Passengers connect to the MQTT broker using a specific passenger application (referred to as a passenger app in this chapter) that is developed on the iOS and Android operating systems. The passenger app embeds an MQTT client that can connect to the MQTT broker and then execute actions using the publish/subscribe paradigm.

- **Back-end system or payment system**: The back-end system is mainly used for payment processing. Actually, it receives payment orders from passengers and processes them. The back-end application also includes an MQTT client that connects to the MQTT broker.

An overview of these actors is shown in Figure 7-1 on page 173.
Figure 7-1  Actors of the PickMeUp solution

Note: Actors in the PickMeUp solution exchange information through MessageSight (the MQTT broker), using the MQTT protocol.

7.2 Stages of the PickMeUp solution

In this section, we describe the stages of the PickMeUp solution. We identify five distinct stages:

- **Connecting**: In this stage, the actors (other than the MQTT broker) connect to MessageSight and publish information related to their identities and times of connection.

- **Pairing**: The purpose of this stage is to link a passenger to a driver. A passenger requests a driver. The driver can view a list of potential passengers, who are waiting for an available driver, and choose a passenger in the list.

- **Approaching**: This stage allows drivers to locate passengers in real time and to recognize them as quickly as possible. Similarly, a passenger needs to locate a driver in real time. Various tools, such as maps, chats (text and voice), and pictures, are proposed by the PickMeUp solution for use in this stage.

- **Riding**: After the driver and passenger are physically located at the same position, the ride starts.
Payment: This last stage starts when the passenger’s destination has been reached. The passenger can then pay for the ride using the PickMeUp application. Moreover, the passenger can rate the driver and tip the driver. The back-end system is in charge of retrieving payment information from passengers and processing the payments.

7.3 Topics and messages for the PickMeUp scenario

We describe the topics and messages that are exchanged between the various actors during the five stages identified in 7.2, “Stages of the PickMeUp solution” on page 173.

For each stage, we describe the quality of service (QoS) level used for publications, the publication type (retained or not retained), and the session type for subscriptions (durable or non-durable). See “Quality of service (QoS) levels and flow” on page 236 for descriptions of the QoS levels.

7.3.1 Connecting

At connection time, three actors connect to the MessageSight server:

- The driver, using the HTML5 PickMeUp driver app.
- The passenger, using either the iOS or Android PickMeUp passenger app.
- The back end, using a dedicated application that is able to integrate with a payment system. The back-end application is based on Node.js.

Each driver, passenger, and the back-end application connect to MessageSight using a unique client identifier. Details about the client identifier are in 6.2, “PickMeUp architecture for real-life use” on page 153.

At the end of the connecting stage, the driver app, passenger app, and back-end application are all connected to the MessageSight server.

Publications at connection time

When the driver app, passenger app, and back-end application are connected, the passenger app and driver app publish a JavaScript Object Notation (JSON) message (we call this a connection message), as shown in Example 7-1.

Example 7-1 Connection message

```json
{
    "name": "driversOrPassengersName",
    "connectionTime": time
}
```

The following details refer to the JSON connection message shown in Example 7-1:

- name: The name of the driver or passenger, as viewed by the passenger or driver
- connectionTime: The connection time in milliseconds, since January, 1st 1970

This JSON message is published as a retained message, using QoS level 0. Each passenger app and driver app publishes the JSON connection message on one of the following dedicated topics:

- Pickmeup/passengers/passengerId for the passenger app
- Pickmeup/drivers/driverId for the driver app
Chapter 7. PickMeUp messaging scenario

Note: In this chapter, passengerId is the client identifier used by the MQTT client of the passenger app to connect to MessageSight. Similarly, driverId is the client identifier used by the MQTT client of the driver app to connect to MessageSight.

Note: The MQTT protocol defines a client identifier (client ID) that uniquely identifies a client on a network. In simple terms, when connecting to a messaging server, a client needs to specify a unique string that is not used currently and will not be used by any other client that will connect to the MQTT server.

Also, to perform the MQTT client connection, both the driver app and the passenger app use the last will and testament property provided by the MQTT protocol, when connecting to MessageSight.

Note: When a client app connects to a messaging server, such as MessageSight, it (the client app) can define a topic and a message that will be published automatically if it unexpectedly disconnects. This is called the last will and testament (LWT).

In the case of the PickMeUp solution, the last will and testament message used by both the driver app and the passenger app is an empty message. This empty message is associated with last will and testament topics, which differ for each driver app and passenger app:

- Pickmeup/passengers/passengerId for the passenger app
- Pickmeup/drivers/driverId for the driver app

Therefore, if the passenger app or driver app is unexpectedly disconnected from the messaging server, the JSON connection message is erased from the MQTT broker (MessageSight) for both the passenger and driver.

When there is a clean disconnection (when the DISCONNECT method (provided by the MQTT protocol) is used to disconnect an MQTT client from an MQTT broker), the two JSON connection messages have to be erased before the passenger app and driver app are disconnected.

Each driver also publishes a picture when connecting to the PickMeUp driver app. Pictures are published using a retained message on the following topic:

Pickmeup/drivers/driverId/picture.

The JSON message used to send a picture is shown in Example 7-2, where base64Binary is the binary content of an image encoded in Base64.

Example 7-2 JSON message used to send a picture

```json
{
  "url" : "data:image/png;base64;base64Binary"
}
```

Subscriptions at connection time

To receive messages from each other, each driver app and passenger app must subscribe to specific and dedicated topics:

- Pickmeup/drivers/driverId/inbox for the driver app
- Pickmeup/passengers/passengerId/inbox for the passenger app

We call these two topics inbox topics.
QoS level 2 is used for subscriptions on inbox topics. This is a guarantee that messages will be received on the inbox topics at the QoS at which they are published during the payment stage (level 2).

The session flag used for subscriptions that use inbox topics is false, which means that the subscription is durable over time. In this case, when a driver or passenger is temporarily disconnected from the MQTT broker because of a network failure, the driver application and the passenger application do not lose any inbox messages that are published on the inbox topics.

A summary of the actors, connections (connect), publications (pub), and subscriptions (sub) at connection time is shown in Figure 7-2.

7.3.2 Pairing

At pairing time, two actors interact with each other:
- Drivers by using the HTML5 PickMeUp driver app
- Passengers by using either the iOS or the Android PickMeUp passenger app

The pairing stage consists of linking an available driver with a passenger who is requesting a ride.
Passenger publications during the pairing stage

A passenger requests a ride by publishing a specific JSON request message on the request topic Pickmeup/requests/passengerId. An example of a JSON request message is shown in Example 7-3.

Example 7-3  Request message from a passenger

```json
{
    "name": "passengersName",
    "lon": longitude,
    "lat": latitude
}
```

The JSON request message includes the passenger name and position in longitude and latitude. This message is published as a retained message. In this way, newly connected drivers can see passenger requests that are pending. The QoS published that the request message is at level 1, acknowledging that the message has been delivered to connected drivers.

Driver subscriptions during the pairing stage

The driver app subscribes to the generic topic Pickmeup/requests/+ and it is therefore able to retrieve passenger requests that are published on the request topic, Pickmeup/requests/passengerId.

The driver app displays a list of passenger requests, including request messages and topics, and passenger names, positions, and identifiers.

Driver publications during the pairing stage

When a driver accepts a ride from the list of requesting passengers, the driver app performs the following actions, in order:

1. It publishes an empty retained message on the topic Pickmeup/requests/passengerId.
   Therefore, the passenger request can be erased from the list of requesting passengers, for all connected driver apps.

2. It publishes a confirmation message to the accepted passenger inbox topic (Pickmeup/passengers/passengerId/inbox), based on the passenger identifier. This message is published using QoS level 1 and is not retained on MessageSight. An example of an acceptance message is shown in Example 7-4.

Example 7-4  Confirmation message from driver

```json
{
    "type": "accept",
    "driverId": driverId,
    "lon": longitude,
    "lat": latitude
}
```

The JSON confirmation message includes an acceptance status that is a static value (accept) and the driver's identifier and position in longitude and latitude.
Passenger subscriptions during the pairing stage

As soon as a passenger receives a confirmation message on the topic \texttt{Pickmeup/passengers/passengerId/inbox}, the passenger app extracts the driver identifier from the JSON confirmation message.

Based on the driver identifier, the passenger app subscribes to the following two topics to obtain the name and position of the driver who accepted the ride, including a picture of the driver:

- \texttt{Pickmeup/drivers/driverId}: A message published on this topic contains the name and position of the driver who is connected to MessageSight and using \textit{driverId} as a client identifier.
- \texttt{Pickmeup/drivers/driverId/picture}: A message published on this topic contains the picture of the driver who is connected to MessageSight and using \textit{driverId} as a client identifier.

A summary of actors, publications (pub), and subscriptions (sub) at pairing time is shown in Figure 7-3 on page 178.

\textbf{Note}: The back-end system does not participate in the pairing stage. Nevertheless, it is always connected to MessageSight from the connecting stage through to the payment stage.
7.3.3 Approaching

During the approaching stage, the PickMeUp solution provides a chat system between the driver app and the passenger app, allowing both people to exchange information.

The aim of this chat is to facilitate the approach for the driver and to reduce waiting time for the passenger. The chat allows the exchange of text and audio data.

**Driver publications during the approaching stage**

During the approaching stage, the driver app publishes messages to the following topics, using QoS level 0:

- Pickmeup/drivers/driverId/location: On this topic, the driver app publishes a positioning message using a JSON format, as shown in Example 7-5.

  **Example 7-5  Positioning message from the driver app**

  ```json
  {
      "lon":longitude,
      "lat":latitude
  }
  ```

  The positioning message provides the longitude and latitude of the driver's position. The positioning message is published as a retained message, so if the passenger app has to reconnect, it obtains the latest driver's position.

- Pickmeup/passengers/passengerId/chat: The driver app publishes text or audio data to the passenger chat topic. Chat messages can include text or audio data. An example chat message that includes text is shown in Example 7-6.

  **Example 7-6  Sample chat message for text**

  ```json
  {
      "format":"text",
      "data":"text"
  }
  ```

  An example of a chat message that includes audio data is shown in Example 7-7.

  **Example 7-7  Sample chat message for audio data**

  ```json
  {
      "format":"data:audio/wav;base64",
      "data":"base64EncodedData"
  }
  ```

  The audio data is encoded in Base64 before being included in the JSON chat message. Chat messages are published as non-retained messages.

**Passenger publications during the approaching stage**

During the approaching stage, the passenger app publishes messages to the following topics, using QoS level 0:

- Pickmeup/passengers/passengerId/location: On this topic, the passenger app publishes a positioning message using a JSON format, as shown in Example 7-5.

  The positioning message is published as a retained message so that, if the driver app has to reconnect, it obtains the latest passenger position.
- Pickmeup/drivers/driverId/chat: The passenger app publishes text or audio data to the passenger chat topic. Chat messages can include text or audio data, as shown in Example 7-6 and Example 7-7.

Chat messages are published as non-retained messages.

During the approaching stage, the passenger app publishes a retained message corresponding to a passenger picture, using QoS level 0. This message is published on topic Pickmeup/passengers/passengerId/picture. The JSON message used to publish a picture is shown in Example 7-2 on page 175.

**Driver subscriptions during the approaching stage**
During the approaching stage, the driver app subscribes to the following topics:

- Pickmeup/passengers/passengerId: The driver app uses this topic to obtain passenger information, as described in “Publications at connection time” on page 174.
- Pickmeup/passengers/passengerId/picture: The driver app uses this topic to obtain the passenger picture, as described in Example 7-2 on page 175.
- Pickmeup/passengers/passengerId/location: The driver app uses this topic to obtain the passenger position in real time.
- Pickmeup/drivers/driverId/chat: The driver app uses this topic to obtain text messages or audio data sent by a passenger.

**Passenger subscriptions during the approaching stage**
During the approaching stage, the passenger app subscribes to the following topics:

- Pickmeup/drivers/driverId/location: The passenger app uses this topic to obtain the driver's position in real time.
- Pickmeup/passengers/passengerId/chat: The passenger app uses this topic to obtain text messages or audio data sent by a driver.

A summary of actors, publications (pub), and subscriptions (sub) at approaching time is shown in Figure 7-4 on page 181.
7.3.4 Riding

The riding stage starts after the approaching stage is completed. This means that a driver and a passenger are located at the same position.

**Driver publications during the riding stage**

At the start of the riding stage, the driver app publishes a message on a passenger inbox topic, `Pickmeup/passengers/passengerId/inbox`, indicating that the trip is starting. We call this message the *start trip message*.

The start trip message has the following characteristics:

- It is based on a JSON format.
- It is published using QoS level 2: This level ensures the deliverance of the start trip message to the passenger.
- It is not retained on MessageSight.

An example of the start trip message is provided in Example 7-8 on page 181.

**Example 7-8  Start trip message**

```json
{
  //...
}
```
The type `tripStart` indicates that the trip has started.

When the trip is completed, the `driver` app publishes a message on a passenger inbox topic, `Pickmeup/passengers/passengerId/inbox`, indicating that the trip has ended. We call this message the `trip-end message`.

The trip-end message has the following characteristics:

- It is based on a JSON format.
- It is published using QoS level 2. This level ensures the deliverance of the trip-end message to the passenger.
- It is not retained on MessageSight.

An example trip-end message is provided in Example 7-9.

```json
{
    "type": "tripEnd",
    "distance": distance,
    "time": time,
    "cost": cost
}
```

The type `tripEnd` indicates that the trip has ended.

The `driver` app embeds specific algorithms, which are used to calculate the cost of the trip based on the time and distance of this trip. An example is shown in Example 9-7 on page 212.

After the trip cost is calculated, the trip-end message is sent to the `passenger` app to notify the passenger of the cost, so that the passenger can proceed with payment.

A summary of the actors, publications (pub), and subscriptions (sub) at riding time is shown in Figure 7-5 on page 183.
7.3.5 Payment

The payment stage starts after the trip ends. The payment is performed by a passenger using the passenger app. The back-end system is part of the payment process.

**Passenger publications during the payment stage**

To process the payment, the passenger app publishes a payment order message on the Pickmeup/payments topic using QoS level 2 to ensure that the message is always delivered and always delivered one time.

**Note:** The message must be stored locally on the passenger app, until the passenger app receives confirmation that the payment order message was published by MessageSight.

Example 7-10 presents an example of a payment order message, which must be published as a not-retained message.

**Example 7-10 Sample payment order message**

```json
{
    "driverId": "driverId",
    "passengerId": "passengerId",
}
```
A payment order message uses a JSON format. A payment order contains the driver’s and passenger’s identifiers. It also contains the cost of the trip, the rating (a one-to-five star rating is used) and the tip given by a passenger.

**Back-end subscriptions during the payment stage**
To receive payment order messages, the back-end application subscribes to MessageSight using the payment topic `Pickmeup/payments`.

The subscription of the back-end system on the payment topic must be done using a QoS level 2 to receive messages on the payment topic at the QoS at which they are published, which is QoS level 2.

**Note:** A publisher is responsible for determining the maximum QoS at which a message can be delivered, but a subscriber is able to downgrade the QoS to one more suitable for its use. The QoS of a message is never upgraded.

For example, if a client has a QoS level 1 subscription to a particular topic, a QoS level 0 message published to that topic is delivered to the client at QoS level 0. A QoS level 2 message published to the same topic is downgraded to QoS level 1 for delivery to the client.

**Back-end publications during the payment stage**
After the payment has been processed by the back-end system, the back-end application informs both the passenger and the driver of the payment status.

The back-end application publishes a *trip processed* message, which is based on a JSON format, as shown in Example 7-11.

**Example 7-11   Sample trip processed message**

```
{  
  "type":"tripProcessed",
  "tip":tip,
  "rating":rating
}
```

The trip processed message type is *tripProcessed*. It contains the tip amount and the driver’s rating that was entered by the current passenger.

The trip processed message is published as non-retained using QoS level 2 on the following inbox topics:
- `Pickmeup/drivers/driverId/inbox`
- `Pickmeup/passengers/passengerId/inbox`

The driver app and passenger app can use this message to confirm that the trip payment is processed.
Chapter 7. PickMeUp messaging scenario

The back-end system also proceeds to clean up the two passenger retained messages: the positioning and the picture message. To clean up these retained messages, an empty message is published on the following topics using QoS level 0:

- **Pickmeup/passengers/passengerId/picture**: This topic corresponds to the passenger picture that was uploaded at the beginning of the approaching stage.
- **Pickmeup/passengers/passengerId/location**: This topic corresponds to the passenger’s last position at the end of the approaching stage.

A summary of the actors, publications (pub), and subscriptions (sub) at payment time is shown in Figure 7-6.

---

### 7.4 Summary of publication and subscription topics for the PickMeUp solution

This section summarizes the publication and subscription topics that are designed for the PickMeUp solution.
7.4.1 Driver app topics

This section describes the driver app topics for both publications and subscriptions.

**Driver publications**

A summary of driver app publications is shown in Table 7-1.

<table>
<thead>
<tr>
<th>Publication topic</th>
<th>Retained</th>
<th>QoS Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickmeup/drivers/driverId</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>Pickmeup/requests/passengerId</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/inbox</td>
<td>No</td>
<td>1,2</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/chat</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/location</td>
<td>Yes</td>
<td>0</td>
</tr>
</tbody>
</table>

**Driver subscriptions**

A summary of driver app subscriptions is shown in Table 7-2.

<table>
<thead>
<tr>
<th>Subscription topics</th>
<th>Session flag</th>
<th>QoS level</th>
<th>Unsubscription needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickmeup/drivers/driverId/inbox</td>
<td>False</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>Pickmeup/requests/+</td>
<td>False</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId</td>
<td>False</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/picture</td>
<td>False</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/location</td>
<td>False</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/chat</td>
<td>False</td>
<td>0</td>
<td>Yes</td>
</tr>
</tbody>
</table>

7.4.2 Passenger app topics

This section describes the passenger app topics for both publications and subscriptions.

**Passenger publications**

A summary of passenger app publications is shown in Table 7-3 on page 186.

<table>
<thead>
<tr>
<th>Publication topic</th>
<th>Retained</th>
<th>QoS Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickmeup/passengers/passengerId</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>Pickmeup/requests/passengerId</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/picture</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/location</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/chat</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>Pickmeup/payments</td>
<td>No</td>
<td>2</td>
</tr>
</tbody>
</table>
Passenger subscriptions
A summary of passenger app subscriptions is shown in Table 7-4.

<table>
<thead>
<tr>
<th>Subscription topics</th>
<th>Session flag</th>
<th>QoS level</th>
<th>Unsubscription needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickmeup/passengers/passengerId/inbox</td>
<td>False</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId</td>
<td>False</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/picture</td>
<td>False</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/chat</td>
<td>False</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>Pickmeup/drivers/driverId/location</td>
<td>False</td>
<td>0</td>
<td>Yes</td>
</tr>
</tbody>
</table>

7.4.3 Back-end application topics
This section describes the back-end application topics for both publications and subscriptions.

Back-end publications
A summary of back-end application publications is shown in Table 7-5.

<table>
<thead>
<tr>
<th>Publication topic</th>
<th>Retained</th>
<th>QoS Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickmeup/drivers/driverId/inbox</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/inbox</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId/location</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>Pickmeup/passengers/passengerId</td>
<td>Yes</td>
<td>0</td>
</tr>
</tbody>
</table>

Back-end subscriptions
A summary of back-end application subscriptions is shown in Table 7-6.

<table>
<thead>
<tr>
<th>Subscription topics</th>
<th>Session flag</th>
<th>QoS level</th>
<th>Unsubscription needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickmeup/payments</td>
<td>False</td>
<td>2</td>
<td>No</td>
</tr>
</tbody>
</table>

We assume the back-end application is always connected to MessageSight, so there is no need for unsubscriptions on the payment topic.
PickMeUp MQTT on iOS

To demonstrate the viability of the various MQTT clients that are available, three separate passenger applications (referred to as passenger apps in this chapter) were developed. In this chapter, we describe the iOS implementation of the PickMeUp iOS mobile application for the passenger and how it makes use of the iOS MQTT client.

We explain the core components of the iOS MQTT client and how they drive the application through the stages described in 7.2, “Stages of the PickMeUp solution” on page 173. We also describe the PickMeUp iOS passenger app. We cover two specific components of the application:

- Text and voice chat features
- XCode storyboards for the user interface

This chapter includes the following topics:

- 8.1, “Advantages of developing a native (iOS) passenger app” on page 190
- 8.2, “Features of the PickMeUp iOS passenger app” on page 192

Note: The iOS MQTT client is distributed as part of the IBM WebSphere MQ Client Pack, which is available at this website:

8.1 Advantages of developing a native (iOS) passenger app

The iOS MQTT client allows developers to create an application that will provide a similar look and feel to other iOS applications and a more familiar user experience. Aside from the similarity in appearance of these applications, using a native client also allows developers to take advantage of features specific to the iOS.

One of the features of the iOS that the PickMeUp passenger app uses is the dispatch queues. The application uses dispatch queues to guarantee serial processing of incoming MQTT messages. When the MQTT client receives a message, the message is passed to the `onMessageArrived:message:` method of the `MqttCallbacks` protocol. The application is then able to move the message onto the dispatch queue that is running on the main thread of the application, so that the client can continue receiving messages.

8.1.1 Using the iOS MQTT client

To use the iOS MQTT client, an application must create an instance of the `MqttClient` class. The `MqttClient` class provides a handle to the client and the APIs of the client. There are five APIs used for communicating with the MQTT broker:

- Connecting to an MQTT broker
- Disconnecting from an MQTT broker
- Subscribing to an MQTT topic filter
- Unsubscribing from an MQTT topic filter
- Publishing MQTT messages

In the PickMeUp iOS passenger app, the `PMUMessenger` class stores the `MqttClient` object as an instance variable.

In addition to the `MqttClient` class, there are three protocols and three structures that are defined in the iOS MQTT client.

iOS MQTT client protocols

The iOS MQTT client defines three protocols that an application can implement:

- `InvocationComplete protocol`

  The `InvocationComplete` protocol specifies the callback methods that the iOS MQTT client uses to notify the application whether an API call is successful. There are two methods for this in the `InvocationComplete` protocol:

  - The `onSuccess:` method is invoked upon successful completion of `MqttClient` API calls.
  - The `onFailure:errorCode:errorMessage:` method is invoked upon failure of `MqttClient` API calls.

  The first argument to each method provides the context for the callback. This is set in the `invocationContext` argument of the API call that resulted in the callback.
The PickMeUp iOS passenger app uses these methods to control the flow of the application. The invocationContext argument will be set to one of the following values:

- `connect`: Set when connecting to the MQTT broker
- `disconnect`: Set when disconnecting from the MQTT broker
- `send`: Set when publishing an MQTT message
- `subscribe/topicFilter`: Set when subscribing to the topic specified by `topicFilter`
- `unsubscribe/topicFilter`: Set when unsubscribing from the topic specified by `topicFilter`

The appropriate action to take is determined based on the content of the invocationContext argument passed into the callback.

**MqttCallbacks protocol**

The `MqttCallbacks` protocol specifies the callback methods that are used when an action occurs that was not initiated by an `MqttClient` API call. For example, `onMessageArrived:message:` is called when a message is published to the iOS MQTT client.

There are three methods in the `MqttCallbacks` protocol:

- The `onMessageArrived:message:` method is invoked upon receipt of an MQTT message.
- The `onMessageDelivered:messageId:` method is invoked upon successful delivery of an MQTT message.
- The `onConnectionLost:errorMessage:` method is invoked if an unexpected disconnect occurs between the client and the MQTT broker.

Similar to the InvocationComplete protocol, the first argument of each of these methods provides the context for the callback.

The PickMeUp iOS passenger app uses these methods for handling events that are received from the MQTT broker. When an MQTT message is passed to the `onMessageArrived:message:` method, the application can parse the topic and data from the message and handle it correctly.

**MqttTraceHandler protocol**

The `MqttTraceHandler` protocol specifies the methods that the iOS MQTT client uses to print trace messages.

There are five severities for trace messages:

- Debug
- Log
- Info
- Warn
- Error

The PickMeUp iOS passenger app does not implement this protocol.

Both the `MqttCallbacks` and InvocationComplete protocol implementations for the PickMeUp application are defined in `PMUCallbacks.m`. 
iOS MQTT client structures

The following structures are defined in the client:

- **ConnectOptions**: The ConnectOptions structure specifies the properties that the client uses to connect to the MQTT broker.
- **SSLOptions**: The SSLOptions structure specifies the properties that the client uses to create a secure connection to the MQTT broker.
- **DisconnectOptions**: The DisconnectOptions structure specifies the properties that the client uses to disconnect from the MQTT broker.

More information about the iOS MQTT client

For more information about how to use the iOS MQTT client, see the documentation included with the MQTT client download.

Note: The iOS MQTT client is distributed as part of the IBM WebSphere MQ Client Pack, which is available at this website:

8.2 Features of the PickMeUp iOS passenger app

We describe the implementation of the PickMeUp iOS passenger app. Specifically, we focus on the chat feature of the application. The chat feature consists of sending and receiving text and audio messages between the passenger app and the driver app.

8.2.1 PickMeUp class overview

Before we explain the implementation of chat messages, it is necessary to understand the classes in the application:

- **PMUAppDelegate**: The PMUAppDelegate class implements the UIApplicationDelegate protocol. The UIApplicationDelegate protocol defines the methods for launching the application, terminating the application, and sending the application to the background. The PMUAppDelegate class also maintains handles to each of the view controllers in the application and an NSMutableArray pointer called messageCache, which stores all chat messages between the passenger and driver.
- **PMUMessenger**: The PMUMessenger class acts as a wrapper around the MqttClient class.
- **PMUDriver**: The PMUDriver class implements the MKAnnotation protocol. The MKAnnotation protocol defines the properties and methods for map view annotations. The PMUDriver class also maintains the details of the driver that is paired with the passenger. These details include the driver ID, name, location, and picture.
- **PMURequest**: The PMURequest class maintains the details of the passenger and the trip. These details include the passenger ID, name, location, picture, trip distance, cost, and time.
- **PMUChatViewController**: The PMUChatViewController displays the PMUMessageTableView.
- **PMUMessageTableView**: The PMUMessageTableView implements the UITableViewDataSource and the UITableViewDelegate protocols, and it is responsible for displaying chat messages and creating chat views.
- **PMUMessage**: The PMUMessage class represents an individual chat message.
PMUCallbacks: PMUCallbacks is not a class itself, but it contains the implementations of both the MqttCallbacks protocol and the InvocationComplete protocol. For the InvocationComplete protocol, a separate implementation exists for each API defined in the MqttClient class.

PMUChatSender: PMUChatSender is an enumerated type that specifies whether the message was sent by the passenger or the driver. There are two values:
- PMUChatPassenger for messages sent by the passenger
- PMUChatDriver for messages sent by the driver

This property is used by PMUChatTableView to determine the positioning of the chat message in the table.

8.2.2 Chat topic structure

The chat feature is implemented using two topics: one for the driver and one for the passenger:
- The driver app subscribes to pickmeup/drivers/driverId/chat to receive incoming chat messages, where driverId is the client identifier of the driver.
- The passenger app subscribes to pickmeup/passengers/passengerId/chat to receive incoming chat messages, where passengerId is the client identifier of the passenger.

In the passenger app, the chat subscription is created during the pairing stage, after a driver has sent an accept status message to the passenger app.

8.2.3 Chat Implementation

In this section, we discuss the implementation for sending, receiving, and displaying chat messages.

Sending chat messages

When the passenger sends a text or audio message, the application creates a JavaScript Object Notation (JSON) formatted string. The following details show the JSON chat message that is published:
- format: the format of the message
  There are two possible formats:
  - text: specifies that the message is a text message
  - data:wav/audio;base64: specifies that the message is an audio message
- data: the content of the message

Example 8-1 shows the JSON string for a text message.

Example 8-1   Text message JSON string

```
{ "format" : "text" , "data" : "This is a sample message." }
```

Example 8-2 shows how a chat message is sent by the passenger app.

Example 8-2   Sample JSON chat message

```swift
/** Construct and send text and audio chat messages. */
-(void)sendMessage:(NSString *)payload
    withFormat:(NSString *)format
```
In Example 8-2 on page 193, the application first constructs the JSON message content. The message is then published by passing it to the `PMUMessenger` `publish:payload:qos:retained:` method. After publishing the message to the driver's chat topic, a `PMUMessage` object is created with the JSON message content. The `PMUChatSender` property of the `PMUMessage` is set to `PMUChatPassenger` to indicate that this message was sent by the passenger. This is used later for determining the position of the message in the chat table view. After creating the `PMUMessage` object, it is passed to the `PMUMessenger displayChatMessage:from:` method.

**Receiving chat messages**

Incoming chat messages are passed to the `PMUCallbacks onMessageArrived:message:` method when they are received by the MQTT client. Example 8-3 shows the callback method for receiving MQTT messages.

**Example 8-3 Callback method for receiving MQTT messages**

```c
/** Process incoming messages. */
-(void)onMessageArrived:(NSObject *)invocationContext
  message:(MqttMessage *)message
{
  NSString *topic = message.destinationName;
  NSArray *topicParts = [[topic componentsSeparatedByString:@"/"]
                 array];
  NSString *payload = [[NSString alloc] initWithBytes:message.payload
            length:message.payloadLength encoding:NSASCIIStringEncoding];
  NSLog(@"MQTT Message Received\n\tTopic: %@\n\tPayload: %@", topic, payload);
  // Convert the message payload to a dictionary for parsing
  NSError* error;
  NSDictionary* json = [NSJSONSerialization
                        JSONObjectWithData:[payload
                         dataUsingEncoding:NSUTF8StringEncoding]
                       error:&error];
  //...
In Example 8-3 on page 194, the application checks the topic of the message to determine the type of message that was received. For chat messages, it checks to determine whether the end of the topic string is chat. If so, the message is passed to the handleChatMessage:topic:payload: method. This method is shown in Example 8-4.

**Example 8-4  Processing an incoming chat message**

```objective-c
/** The incoming message is a chat message. */
-(void)handleChatMessage:(NSDictionary *)json topic:(NSString *)topic payload:(NSString *)payload
{
    PMUMessenger *messenger = [PMUMessenger sharedMessenger];
```
// parse chat message (text or audio)
NSString* format = [json objectForKey:PMUFormatField];
PMUChatSender sender = PMUChatDriver;

// Use timestamp based on receiving clients time, so that messages show up in
// the order
// that they arrive.
NSDate* date = [[[NSDate alloc] initWithTimeIntervalSince1970:[[NSDate date]
timeIntervalSince1970]]];
PMUMessage *message = [PMUMessage dataWithText:payload date:date sender:sender
format:format];
[messenger displayChatMessage:message from:[PMUTopicFactory getDriverId]];
}

In Example 8-4 on page 195, the application parses the JSON string and uses the content to
create a PMUMessage object. All chat messages received from the driver have their
PMUChatSender property set to PMUChatDriver, which is used later for determining the
placement of the message in the chat table view. After creating the PMUMessage object, it is
passed to the PMUMessenger displayChatMessage:from: method.

Displaying chat messages
In both sending and receiving messages, PMUMessenger displayChatMessage:from: is
called to display the message in the chat table view of the application. Example 8-5 shows the
method for displaying a chat message.

Example 8-5  Adding a chat message to the table
/** An incoming chat message arrived. Add it to the message cache,
* and play the message if it was sent by the driver.
*/
-(void)displayChatMessage:(PMUMessage *)message
from:(NSString *)senderId
{
    PMUAppDelegate *appDelegate = [[UIApplication sharedApplication] delegate];
    // The message always goes into the cache for the chat view.
    [appDelegate.messageCache addObject:message];
    [appDelegate.chatController reload];
    if ([senderId isEqualToString:[PMUTopicFactory getPassengerId]] == NO)
    {
        // Only autoplay message if it came from the driver.
        [appDelegate.chatController playMessageOnArrival:message];
    }
}

In Example 8-5, the message is added to the messageCache array of the PMUAppDelegate
object. The chat table view is then reloaded to display the newly added chat message. If the
new message was sent by the driver and it is an audio message, the message is played
automatically upon receipt.

Finally, the chat message is drawn in the chat table. This is performed by the
UITableViewDelegate tableView:cellForRowAtIndexPath: method, as shown in
Example 8-6 on page 197.
Example 8-6  Drawing a chat message in the table

```objective-c
/** Draw the table cell for a chat message. */
- (UITableViewCell *)tableView:(UITableView *)tableView
tableView:cellForRowAtIndexPath:(NSIndexPath *)indexPath
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [[UITableViewCell alloc] init];
cell.selectionStyle = UITableViewCellSelectionStyleNone;
PMUMessage *data = [self.chatSection objectAtIndex:indexPath.row];

    // Create the subviews for the table cell
    UIImageView *avatarImage = [self createAvatarImageSubview:data.type];
    [cell addSubview:avatarImage];
    UIImageView *bubbleImage = [self createBubbleImageSubview:data];
    [cell addSubview:bubbleImage];
    UIView *textView = [self createTextView:data];
    [cell addSubview:textView];

    if ([data.format isEqualToString:PMUAudioFormat])
    {
    UIButton *voiceButton = [self createAudioButtonSubview:bubbleImage];
    [cell addSubview:voiceButton];
    }

    [cell setBackgroundColor:[UIColor colorWithRed:58/255.0 green:74/255.0
        blue:83/255.0 alpha:1.0]];

    return cell;
}
```

Example 8-6 shows the method for drawing each table cell. Each cell corresponds to a
separate chat message. The messageCache array of PMUAppDelegate is used as the data
source for the chat table. For each PMUMessage object contained in messageCache, the
tableView:cellForRowAtIndexPath: method is called. The method gets a chat message from
messageCache using indexPath.row as the array index for accessing messageCache. Several
views are then created and added to the cell:

- UIImageView avatarImage: Displays the picture of the sender of the message. For the
driver, this will appear to the left of the message content. For the passenger, this will
appear to the right of the message content.

- UIImageView bubbleImage: The background image for the message content.

- UITextView textView: The view of the text within the chat bubble.

The frame for the avatar image is a fixed size. The frames for the other views are set
dynamically, based on the message content. The positioning of the chat message in the chat
table view depends on the value of the PMUChatSender property of the message. If the value
of the PMUChatSender property is PMUChatDriver, the message is positioned on the left side
of the chat table. If the value of the PMUChatSender property is PMUChatPassenger, the message
is positioned on the right side of the chat table. Figure 8-1 on page 198 displays the views of
the chat cells as they appear in the application.
8.2.4 The PickMeUp iOS passenger app storyboard

The PickMeUp iOS passenger app uses three storyboards. A storyboard defines the layout of the user interface for the application. It also displays the connections among the elements of the user interface and the matching class or property in the source code.

---

1 All images on this page were taken using the iOS Simulator application from Apple Inc.
Three storyboard files are used:

- **Main_iPhone.storyboard**: Defines the user interface for a 4-inch Apple iPhone. A sample from the storyboard is shown in Figure 8-2.

![Sample of a Main_iPhone.storyboard](image)

*Figure 8-2  Sample of a Main_iPhone.storyboard*

---

2 All images on this page were taken using the iOS Simulator application from Apple Inc.
Secondary_iPhone.storyboard: Defines the user interface for a 3.5-inch iPhone. A sample from the storyboard is shown in Figure 8-3.

Figure 8-3 Sample of a Secondary_iPhone.storyboard

---

3 All images on this page were taken using the iOS Simulator application from Apple Inc.
Main_iPad.storyboard: Defines the user interface for an iPad. A sample from the storyboard is shown in Figure 8-4.

![Sample of a Main_iPad.storyboard](image)

Figure 8-4  Sample of a Main_iPad.storyboard

Depending on the type of device on which the application is being run, it loads the appropriate storyboard. This is done so that the application fits correctly on each of the various devices. An alternative is to use constraints to define how the different elements of the storyboard move and resize, based on the dimensions of the device.

---

4 All images on this page were taken using the iOS Simulator application from Apple Inc.
PickMeUp MQTT on Android

To demonstrate the viability of the various MQTT clients that are available, three separate passenger applications (referred to as passenger apps in this chapter) were developed. In this chapter, we describe the Android implementation of the PickMeUp iOS mobile application for the passenger app.

This chapter includes the following topics:

- 9.1, “Advantages of developing an Android PickMeUp application” on page 204
- 9.2, “Features of the Android PickMeUp application” on page 204
9.1 Advantages of developing an Android PickMeUp application

The Eclipse Paho project Android Service is an open source interface to the Eclipse Paho Java MQTT client library. However, it is important to mention that any Android Service can be modified or implemented to handle MQTT messaging using the Eclipse Paho Java MQTT client library.

Note: The Java MQTT client and the Android Service are both distributed as part of the open source Eclipse Paho project and are available at this website:
http://www.eclipse.org/paho/

One of the most important benefits of developing MQTT applications on Android is the capability of the Android OS to execute long-running operations in the background, without requiring any user input. By wrapping all of the messaging logic inside an Android Service that continuously runs in the background, even when the application is not active, it is possible to perform the following functions:

- Ensure that the application messaging engine is always active and can receive and process messages, regardless of the application, OS, or device state. This capability includes waking up the CPU to send a keep alive message.
- Offload message handling to a background thread by using a separate thread for the Android Service.
- Minimize the resource-intensive task of connecting to the MQTT broker by keeping a long-running, open session.
- Provide a clean separation of the application logic from the messaging engine.

The Eclipse Paho open source project (introduced in 1.2.4, “The Eclipse Paho project” on page 9) provides an example implementation of the Android MQTT Service. The Eclipse Paho Android Service is a simple-to-use interface to the Eclipse Paho Java MQTT client. It is sufficient for an introduction to the MQTT messaging patterns in the Android application. If the project requires a more low-level control of the messaging engine inside the application, consider creating a project-specific version of Android Service that will take full advantage of the Eclipse Paho Java MQTT client.

9.2 Features of the Android PickMeUp application

To demonstrate the viability of the various MQTT clients that are available, three separate passenger applications (referred to as passenger apps in this chapter) were developed. In this chapter, we describe the Android PickMeUp application. The other passenger apps are described in Chapter 8, “PickMeUp MQTT on iOS” on page 189 and in Chapter 10, “PickMeUp MQTT in HTML5 applications” on page 217.
In this chapter, we focus on the following topics:

- The Android implementation of the PickMeUp passenger app and how it uses the Eclipse Paho Java MQTT client library. See 9.1, “Advantages of developing an Android PickMeUp application” on page 204.

- The implementation of the Eclipse Paho Android Service for asynchronous handling of sending and receiving messages using the MQTT protocol. See 9.2.2, “Using the Eclipse Paho Android service” on page 208.

- The messaging part of the application and the interactions between the MQTT service and the UI elements of the application. See 9.2.4, “Payment message flow” on page 209.

- The flow of an incoming MQTT message through the Android application using an example during the PickMeUp payment phase. See 9.2.4, “Payment message flow” on page 209.

The Eclipse Paho Android Service is an open source interface to the Eclipse Paho Java MQTT client library; however, it is important to mention that any Android Service can be modified or implemented to handle MQTT messaging using the Eclipse Paho Java MQTT client library.

**Note:** The Eclipse Paho Java MQTT client and Eclipse Paho Android Service are both distributed as part of the open source Paho project and are available at this website:

http://www.eclipse.org/paho/

The Eclipse Paho open source project (introduced in 1.2.4, “The Eclipse Paho project” on page 9) provides an example implementation of the Android MQTT Service. The Eclipse Paho Android Service is a simple-to-use interface to the Eclipse Paho Java MQTT client, and it is sufficient for an introduction to the MQTT messaging patterns in the Android application. If the project requires a more low-level control of the messaging engine inside the application, consider creating a project-specific version of Android service that will take full advantage of the Eclipse Paho Java MQTT Client.

### 9.2.1 PickMeUp class overview

To begin the PickMeUp class overview, it is important to understand the main building blocks of the PickMeUp Android application, as depicted in Figure 9-1 on page 206.
The PickMeUp application on Android can be separated into four main blocks:

- **Android MQTT Service**
  
The Android MQTT Service is an interface to the Eclipse Paho MQTT client library that provides a long-running background service for handling the sending and receiving MQTT messages.

- **PickMeUp Util Classes**
  
The PickMeUp Util Classes are utility classes for a wide range of functions, including facilitating the interaction between activities and the MQTT service and handling incoming MQTT messages. The following classes are the most important:

  - **MqttHandler**
    
    MqttHandler is a utility class that holds an instance of the MqttAndroidClient class and interacts with the MQTT service. This helper class performs all of the basic operations on the MqttAndroidClient, such as connect, disconnect, subscribe, unsubscribe, and publish.

    MqttHandler implements MqttCallback, which allows it to process incoming messages from the MQTT service. MqttHandler does not hold any application logic that processes the incoming messages. That logic is offloaded to the MessageConductor utility class.
– **ActionListener**

*ActionListener* is a utility class that is instantiated every time an Android MQTT client API is used. *ActionListener* implements the IMqttActionListener class to notify the application if the API call was successful.

The Android PickMeUp passenger app uses *ActionListener* methods to control the flow of the application. When `onSuccess(...)` is called for a successful connection, the application then creates the subscription to the passenger inbox topic. When `onSuccess(...)` is called for successfully subscribing to the inbox topic, the application begins publishing its presence, request, photo, and location messages.

– **MessageFactory**

*MessageFactory* is a simple helper class that constructs messages in the format expected by the PickMeUp driver application (referred to as the *driver app* in this chapter). The messaging patterns implemented in the PickMeUp application require that all messages are formatted as JavaScript Object Notation (JSON). *MessageFactory* always returns a JSON-formatted string that can be used directly as a message attribute inside the `publish(...)` method.

– **TopicFactory**

*TopicFactory* is a convenience class that constructs required messaging topics from the application constants and the driver and passenger variables.

– **MessageConductor**

*MessageConductor* holds all of the application logic for routing the incoming MQTT message to the correct area of the application. By inspecting the incoming MQTT message topic and payload, *MessageConductor* creates an intent with the message payload data and broadcasts it to a *RouterReceiver* or a specific broadcast receiver that is registered directly with an activity.

For convenience, all of the utility classes are grouped in the `com.ibm.PickMeUp.utils` package.

– **Broadcasts**

*Broadcasts* is a group of BroadcastReceivers that facilitates communications between activities and other parts of the application:

– **RouterReceiver**

*RouterReceiver* is a BroadcastReceiver that is registered directly with the PickMeUp application at the manifest level. The role of the *RouterReceiver* is to catch broadcasts from the *MessageConductor*, and then process them by performing one of the following actions:

  - Starting a new activity
  - Pushing the incoming message to another utility class for processing
  - Rethrowing the broadcast if it needs to be handled directly by an activity

– **PickMeUp activities**

*PickMeUp* activities is a group of application components that provides graphical interface and capabilities for user interaction. Each *PickMeUp* activity extends Android activity and uses the layout defined within the XML layout files that are inside the application resources. Several of the activities register BroadcastReceivers to interact with other parts of the application through broadcasted messages.

For convenience, all the activities are grouped under the `com.ibm.PickMeUp.activities` package.
9.2.2 Using the Eclipse Paho Android service

To use the Eclipse Paho Android Service, perform the following steps:

1. Add org.eclipse.paho.client.mqttv3.jar to your Android project.
2. Add org.eclipse.paho.android.service.jar to your Android project.
3. Update your application with a reference to the Service class by modifying the AndroidManifest.xml file to include the following line:
   `<service android:name="org.eclipse.paho.android.service.MqttService" />
4. Add the following permission to the AndroidManifest.xml file:
   `<uses-permission android:name="android.permission.WAKE_LOCK" />
5. Create an instance of the MqttAndroidClient class. This class is the application handle to the client, and it provides APIs for the following tasks:
   - Connecting to an MQTT broker
   - Disconnecting from an MQTT broker
   - Subscribing to an MQTT topic filter
   - Unsubscribing from an MQTT topic filter
   - Publishing MQTT messages
   In the Android PickMeUp passenger app, the MqttHandler class stores the MqttAndroidClient object as an instance variable.
6. Create a class to implement the MqttCallback interface.
   In the Android PickMeUp passenger app, the MqttHandler class implements the MqttCallback interface and serves as a single point of contact for all actions related to the MQTT service.
7. Create a class that will implement the IMqttActionListener interface.
   In the Android PickMeUp passenger app, the ActionListener class implements the IMqttActionListener interface and handles receiving information from the MqttAndroidClient client.

Android MQTT client interfaces
There are two client interfaces that need to be implemented to successfully use the Eclipse Paho Android Service:

▶ The IMqttActionListener interface

The IMqttActionListener interface specifies the callback methods that the MqttAndroidClient uses to notify the application whether an API call was successful. There are two methods in the IMqttActionListener interface:

- The onSuccess(IMqttToken token) method is invoked upon the successful completion of MqttAndroidClient API calls.
- The onFailure(IMqttToken token, Throwable throwable) method is invoked upon the failure of MqttAndroidClient API calls.

For the Android PickMeUp passenger app, the IMqttActionListener interface is implemented by the ActionListener class.
Chapter 9. PickMeUp MQTT on Android

The MqttCallback interface

The MqttCallback interface specifies the callback methods that are used when an action occurs that was not initiated by an MqttAndroidClient API call. For example, messageArrived(String topic, MqttMessage message) is called when a message is published to the Android MQTT client.

There are three methods in the MqttCallback interface:

- The messageArrived(String topic, MqttMessage message) method is invoked upon the receipt of an MQTT message.
- The deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) method is invoked upon the successful delivery of an MQTT message.
- The connectionLost(Throwable throwable) method is invoked if an unexpected disconnect occurs between the client and the MQTT broker.

The Android PickMeUp passenger app uses these methods for handling events that are received from the MQTT broker. When an MQTT message is passed to the messageArrived(...) method, depending on the topic and the message payload, the application is then able to handle the message correctly.

For the Android PickMeUp passenger app, the MqttCallback interface is implemented by the MqttHandler class.

9.2.3 More information about the Android MQTT Client

For more information about how to use the Android MQTT client, see the documentation that is included with the Eclipse Paho Android Service download at this website:

http://www.eclipse.org/paho/

9.2.4 Payment message flow

At the end of the PickMeUp trip, the driver sends a message to the passenger inbox topic with the trip details that are displayed on the passenger TripEndDetailsActivity. This message with type set to tripEnd triggers the passenger app to change windows from DriverArrivedActivity to TripEndDetailsActivity.

Example 9-1 shows the passenger inbox topic used in this scenario.

**Example 9-1   Passenger inbox topic**

Pickmeup/passengers/<passengerId>/inbox

Example 9-2 shows the payload of the message that is sent to the passenger inbox topic after the trip ends.

**Example 9-2   End of trip message payload**

{"type": "tripEnd", "distance": "14.20", "time": 13, "cost": "15.76"}

Example 9-3 on page 210 shows the message as received by the Android MQTT Service that is sent to the messageArrived(String topic, MqttMessage mqttMessage) method inside the PickMeUp MqttHandler.
Example 9-3  Implementation of the messageArrived(...) method

```java
@Override
public void messageArrived(String topic, MqttMessage mqttMessage) throws Exception {
    Log.d(TAG, "messageArrived() entered");
    String payload = new String(mqttMessage.getPayload());
    Log.d(TAG, "messageArrived - Message received on topic " + topic
            + ": message is " + payload);
    try {
        // send the message through the application logic
        MessageConductor.getInstance(context).steerMessage(payload, topic);
    } catch (JSONException e) {
        Log.e(TAG, "messageArrived() - Exception caught while steering a message", e.getCause());
        e.printStackTrace();
    }
}
```

Example 9-4 shows an implementation of the messageArrived(...) method inside the MqttHandler class. The processing of the message is offloaded to a MessageConductor utility class.

Example 9-4  steerMessage(...) method to apply the application logic to the incoming MQTT message

```java
public void steerMessage(String payload, String topic) throws JSONException {
    Log.d(TAG, "steerMessage() entered");
    // create a JSONObject from the payload string
    JSONObject jsonPayload = new JSONObject(payload);
    After a JSONObject is created from the MQTT message payload, it runs through the application logic to determine which message the application is dealing with. Then, the JSONObject sends a broadcast message, depending on the situation. For a tripEnd message, the code shown in Example 9-5 is executed.

Example 9-5  Steering a tripEnd message

```java
} else if (topic.equals(TopicFactory.getInstance(context).getPassengerInboxTopic()) &&
            jsonPayload.has(Constants.TYPE)) {
    if (jsonPayload.get(Constants.TYPE).equals(Constants.TRIP_START)) {
        // trip started message - send it to the router
    } else if (jsonPayload.get(Constants.TYPE).equals(Constants.TRIP_END) &&
               jsonPayload.has(Constants.TIME) &&
               jsonPayload.has(Constants.COST) &&
               jsonPayload.has(Constants.DISTANCE)) {
```
In Example 9-5 on page 210, we see that the steerMessage(...) method inspects the topic and the payload to match them to a passenger inbox topic and the tripEnd message type. In addition, there is a check that the payload has the time, cost, and distance values to avoid an exception when attempting to obtain those values from the payload.

A new intent is created with the action attribute set to the same value as the intent filter for the RouterReceiver broadcast receiver. To eliminate the need for the RouterReceiver to go through the complex logic of determining the origin and the destination of the incoming message, an internal flag ROUTE_MESSAGE_TYPE is used. In addition, all of the important parts of the message payload are added to the intent before the intent is broadcasted.

The broadcast is caught and processed by the RouterReceiver.

Example 9-6 RouterReceiver example of BroadcastReceiver processing incoming MQTT messages

```java
public class RouterReceiver extends BroadcastReceiver {
    private final static String TAG = RouterReceiver.class.getName();

    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "onReceive() entered");

        // route the message depending on the ROUTE_MESSAGE_TYPE value
        if (intent.getStringExtra(Constants.ROUTE_MESSAGE_TYPE).equals(Constants.ACTION_INTENT_DRIVER_DETAILS_RECEIVED)) {
            // driver details received
            driverDetailsReceived(context, intent);
        } else if (intent.getStringExtra(Constants.ROUTE_MESSAGE_TYPE).equals(Constants.ACTION_INTENT_START_TRIP)) {
            // trip started message - collect time, distance, cost and send to the router
            Intent actionIntent = new Intent(Constants.ACTION_INTENT_ROUTE_MESSAGE);
            actionIntent.putExtra(Constants.ROUTE_MESSAGE_TYPE, Constants.ACTION_INTENT_START_TRIP);
            String time = jsonPayload.getString(Constants.TIME);
            String distance = jsonPayload.getString(Constants.DISTANCE);
            String cost = jsonPayload.getString(Constants.COST);
            actionIntent.putExtra(Constants.TIME, time);
            actionIntent.putExtra(Constants.DISTANCE, distance);
            actionIntent.putExtra(Constants.COST, cost);
            context.sendBroadcast(actionIntent);
        } else if (intent.getStringExtra(Constants.ROUTE_MESSAGE_TYPE).equals(Constants.ACTION_INTENT_END_TRIP)) {
            // trip ended message - collect time, distance, cost and send to the router
            Intent actionIntent = new Intent(Constants.ACTION_INTENT_ROUTE_MESSAGE);
            actionIntent.putExtra(Constants.ROUTE_MESSAGE_TYPE, Constants.ACTION_INTENT_END_TRIP);
            String time = jsonPayload.getString(Constants.TIME);
            String distance = jsonPayload.getString(Constants.DISTANCE);
            String cost = jsonPayload.getString(Constants.COST);
            actionIntent.putExtra(Constants.TIME, time);
            actionIntent.putExtra(Constants.DISTANCE, distance);
            actionIntent.putExtra(Constants.COST, cost);
            context.sendBroadcast(actionIntent);
        } else if (jsonPayload.get(Constants.TYPE).equals(Constants.TRIP_PROCESSED)) {
            // payment processed message - send it directly to the waiting activity
            ........................................................
            ........................................................
        }
    }
}
```
// start trip message received
startTripMessageReceived(context, intent);
} else if
(intent.getStringExtra(Constants.ROUTE_MESSAGE_TYPE).equals(Constants.ACTION_INTENT_END_TRIP)) {
  // end trip message received
  endTripMessageReceived(context, intent);
} else if
(intent.getStringExtra(Constants.ROUTE_MESSAGE_TYPE).equals(Constants.ACTION_INTENT_CHAT_MESSAGE_RECEIVED)) {
  // chat message received
  chatMessageReceived(context, intent);
} else if
(intent.getStringExtra(Constants.ROUTE_MESSAGE_TYPE).equals(Constants.ACTION_INTENT_DRIVER_ACCEPTED)) {
  // driver accepted passenger message received
  driverAcceptedMessageReceived(context, intent);
}

The RouterReceiver is designed to take action on most of the messages that come through the MQTT service. The logical split between the RouterReceiver and the MessageConductor is for the MessageConductor to understand what the incoming message relates to and which data needs to be extracted from it, and for the RouterReceiver to take action using the incoming message data.

**Example 9-7  Example of RouterReceiver processing an endTrip message**

```java
private void endTripMessageReceived(Context context, Intent intent) {
  Log.d(TAG, "endTripMessageReceived() entered");

  // call the TripEndDetailsActivity - setting FLAG_ACTIVITY_NEW_TASK because we're
  // calling the activity from a BroadcastReceiver
  Intent tripEndIntent = new Intent(context, TripEndDetailsActivity.class);
  tripEndIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
  tripEndIntent.putExtra(Constants.COST, intent.getStringExtra(Constants.COST));
  tripEndIntent.putExtra(Constants.DISTANCE, intent.getStringExtra(Constants.DISTANCE));
  tripEndIntent.putExtra(Constants.TIME, intent.getStringExtra(Constants.TIME));
  context.startActivity(tripEndIntent);
}
```

After the RouterReceiver receives the endTrip message, a new activity is started, which is named the TripEndDetailsActivity. This activity displays the details about the trip, such as the cost, distance traveled, and the time. It also provides a method for the passenger to rate the driver and add a tip for services, as demonstrated in Figure 9-2 on page 213.
Figure 9-2  Trip Details window that follows the received tripEnd message

After the passenger clicks the Submit Payment button (Figure 9-2), the next messaging cycle is initiated. The trip cost, driver rating, and tip are added to the intent that is used to start the PaymentSentActivity. As shown in Example 9-8, the PaymentSentActivity is preparing to submit the payment message and publish it to the payment topic.

Example 9-8  Publishing the payment message

```java
// get trip cost, total payment, rating and the tip passed inside the startActivity intent
String cost = getIntent().getStringExtra(Constants.COST);
String tip = getIntent().getStringExtra(Constants.TIP);
String rating = getIntent().getStringExtra(Constants.RATING);

// get hold of the utils
MqttHandler mqttHandler = MqttHandler.getInstance(this);
TopicFactory topicFactory = TopicFactory.getInstance(this);
MessageFactory messageFactory = MessageFactory.getInstance(this);

// publishing payment message
mqttHandler.publish(topicFactory.getPassengerPaymentTopic(),
    messageFactory.getPaymentMessage(cost, tip, rating), false, 2);
```
As shown in Example 9-8 on page 213, three utility classes are used to publish a message: TopicFactory, MessageFactory, and MqttHandler. Each utility class is described.

**TopicFactory**

TopicFactory getPassengerPaymentTopic() returns a topic string to be used when publishing the message (Example 9-9).

Example 9-9  Getting the passenger payment topic

```java
public String getPassengerPaymentTopic() {
    return Constants.PICK_ME_UP + Constants.PAYMENTS;
}
```

The returned topic string is shown in Example 9-10.

Example 9-10  Passenger payment topic example

```
Pickmeup/payments
```

**MessageFactory**

MessageFactory getPaymentMessage(String cost, String tip, String rating) is used to generate a JSON formatted string for the message payload, as shown in Example 9-11.

Example 9-11  Generating JSON formatted payload for the payment message

```java
public String getPaymentMessage(String cost, String tip, String rating) {
    Log.d(TAG, ".getPaymentMessage() entered");
    JSONObject msg = new JSONObject();
    try {
        msg.put(Constants.COST, cost);
        msg.put(Constants.TIP, tip);
        msg.put(Constants.RATING, rating);
        msg.put(Constants.DRIVER_ID, app.getDriverId());
        msg.put(Constants.PASSENGER_ID, getPassengerName());
    } catch (JSONException e) {
        Log.d(TAG, ".getPaymentMessage() - Exception caught while generating a JSON object", e.getCause());
    }
    return msg.toString();
}
```

In Example 9-11, a JSONObject is created, and all of the relevant message details are added to the object before returning its string representation. The universally unique identifiers (UUIDs), one for the driver app and one for the passenger app, are added to the message payload, as shown in Example 9-12. The Driver ID is taken from the PickMeUpApplication class where the UUID is stored in memory as a globally shared variable. The Passenger ID is taken from the Android Shared Preferences, where it is stored during the login stage.

Example 9-12  String representation of the JSON object returned from the getPaymentMessage(…)

```
{"rating":"0","cost":"17.6","passengerId":"Bob","tip":"0","driverId":"Jessie"}
```
MqttHandler

MqttHandler publish(String topic, String message, boolean retained, int qos) is used to publish the message to the topic, as shown in Example 9-13.

Example 9-13  Publish a message to the topic

```java
public void publish(String topic, String message, boolean retained, int qos) {
    Log.d(TAG, "publish() entered");

    // check if client is connected
    if (!isMqttConnected()) {
        return;
    }

    // create a new MqttMessage from the message string
    MqttMessage mqttMsg = new MqttMessage(message.getBytes());
    // set retained flag
    mqttMsg.setRetained(retained);
    // set quality of service
    mqttMsg.setQos(qos);

    try {
        // create ActionListener to handle message published results
        ActionListener listener = new ActionListener(context, Constants.ActionStateStatus.PUBLISH);
        Log.d(TAG, "publish() - Publishing " + message + " to: " + topic + ", with QoS: " + qos + " with retained flag set to " + retained);
        client.publish(topic, mqttMsg, context, listener);
    } catch (MqttPersistenceException e) {
        Log.e(TAG, "MqttPersistenceException caught while attempting to publish a message", e.getCause());
    } catch (MqttException e) {
        Log.e(TAG, "MqttException caught while attempting to publish a message", e.getCause());
    } else {
        handleMqttDisconnected();
    }
}
```

To publish the message, a new MqttMessage is created from the message string and retained flag. The quality of service (QoS) is set on the message, too. An instance of ActionListener is created to track whether the publish API call was successful. The publish method is then called on an instance of MqttAndroidClient.
PickMeUp MQTT in HTML5 applications

To demonstrate the viability of the various MQTT clients that are available, the passenger and driver applications (referred to as the passenger app and the driver app) were developed for three separate platforms. In this chapter, we describe the HTML5 implementation of the PickMeUp mobile application for the driver app.

This chapter includes the following topics:

- 10.1, “Advantages of developing an HTML5 PickMeUp application” on page 218
- 10.2, “Features of the HTML5 MQTT application” on page 218
10.1 Advantages of developing an HTML5 PickMeUp application

Developing an MQTT-based application with HTML5 increases the portability of the application. Not only can the application be included in a hybrid application framework, such as IBM Worklight® or Apache Cordova, for wider target distribution, but the application can be run from any machine with a supported web browser. The Eclipse Paho project contains a JavaScript MQTT client that can be used to extend MQTT messaging to web browsers and hybrid applications. The PickMeUp driver app uses the Paho JavaScript MQTT client to provide an application suitable for use from a tablet device residing in a driver's vehicle.

The JavaScript MQTT client connects to an MQTT broker over the HTML5 websockets transport. Therefore, a browser supporting the HTML5 websockets specification must be used. The following minimum browser versions are required to run the MQTT client for several common browsers:

- Internet Explorer 10+
- Firefox 21+
- Chrome 21+
- Safari 6+
- Opera 12.1+
- iOS (Safari) 6+
- Android OS 4.4+

10.2 Features of the HTML5 MQTT application

The JavaScript MQTT client has a simple application programming interface (API) to connect, subscribe to topics, publish messages, and receive messages. However, the management of topics, subscriptions, and callbacks can quickly grow complex without a good framework to manage the MQTT client. In this section, we present Messenger.js, which is a framework that is used to manage MQTT messaging for the PickMeUp application.

10.2.1 PickMeUp Messenger overview

The Messenger.js framework abstracts all of the MQTT messaging for the driver session into a unique namespace. Four management objects are defined in the Messenger namespace:

- **TopicManager**: The TopicManager object is used to manage application topics. This object allows for a clean separation between the topic string used for subscriptions and publications and the meaning of the topic. The TopicManager functions are used by the MessageFactory and SubscriptionFactory management objects, not by the driver application session code. Example 10-1 on page 219 shows the TopicManager function used to obtain the topic for passenger location data.
Example 10-1  How TopicManager obtains a passenger location topic

```javascript
var TopicManager = (function() {
    var _basePassengerTopic = "pickmeup/passengers/";
    var _passengerId = null;  // will be set
    ...
    var getPassengerLocationTopic = function() {
        return _basePassengerTopic + _passengerId + "/location";
    }
    ...
    return {
        getPassengerLocationTopic: getPassengerLocationTopic,
        ...
    }();
})();
```

*MessageFactory:* MessageFactory is a factory object, with functions that build an MQTT message based on application parameters. MessageFactory is used to logically organize MQTT message creation into a single set of functions, and hide details, such as topics, quality of service (QoS), and retain flags from the user. Example 10-2 shows a function of MessageFactory that builds a driver location MQTT message object from the driver’s GPS position, using the TopicManager.

Example 10-2  How MessageFactory creates a driver location message

```javascript
var MessageFactory = (function() {
    ...
    var getDriverLocationMessage = function(driverGeo) {
        return {
            topic: TopicManager.getDriverLocationTopic(),
            payload: JSON.stringify({
                lon: driverGeo.lon,
                lat: driverGeo.lat
            }),
            qos: 0,
            retained: true
        }
    }
    ...
    return {
        getDriverLocationMessage: getDriverLocationMessage,
        ...
    }();
});
```

*SubscriptionFactory:* Like MessageFactory, SubscriptionFactory is a factory object with functions that build MQTT subscription data. SubscriptionFactory is used to logically organize subscription definitions for an application, and abstract the topic and QoS from the driver application. Example 10-3 on page 220 shows a function of the SubscriptionFactory that builds and returns subscription details for the passenger location, using the TopicManager.
Example 10-3  How SubscriptionFactory creates a passenger location subscription definition

```javascript
var SubscriptionFactory = (function() {
    var getPassengerLocationSubscription = function() {
        var topic = TopicManager.getPassengerLocationTopic();
        return {
            topic: topic,
            qos: 0,
            onSuccess: function() { if (Utils.TRACE) {
                console.log("subscribed to " + topic); }
            }
        }
    }
    return {
        getPassengerLocationSubscription: getPassengerLocationSubscription,
    }
})();
```

**MessageHandler**: MessageHandler is responsible for handling messages received by the MQTT client. When an MQTT message is received, MessageHandler first checks the topic string with a general handler function, `processMessage`, to determine which specific handler to call. The called handler then parses the JSON payload of the MQTT message and passes data to the appropriate driver application function. Example 10-4 shows the general handler function definition and specific handler for processing passenger location MQTT messages.

Example 10-4  How MessageHandler processes a passenger location message

```javascript
var MessageHandler = (function() {
    var processMessage = function(topic, payload) {
        if (topic.match(TopicManager.getPassengerLocationTopic())) {
            _processPassengerLocationMessage(topic, payload);
        }
    }
    return {
        processMessage: processMessage
    }
})();
```
The Messenger namespace contains functions to connect, disconnect, publish, and subscribe. These functions are called by the driver application and use the four management objects (TopicManager, MessageFactory, SubscriptionFactory, and MessageHandler) to build the appropriate MQTT messages and subscription data. Example 10-5 shows the Messenger namespace publish and subscribe functions, which use objects created by the MessageFactory and SubscriptionFactory to call the JavaScript MQTT client publish and subscribe methods.

Example 10-5   Messenger publish and subscribe

```javascript
var Messenger = (function(global) {
  var TopicManager = (function() {
    /*
    */
  })();
  var MessageFactory = (function() {
    /*
    */
  })();
  var SubscriptionFactory = (function() {
    /*
    */
  })();
  var MessageHandler = (function() {
    /*
    */
  })();

  /*
  */
  var _client = new Messaging.Client(SERVER, PORT, CLIENT_ID);

  var publish = function(msgFactoryObj) {
    var topic = msgFactoryObj.topic;
    var payload = msgFactoryObj.payload;
    var qos = msgFactoryObj.qos;
    var retained = msgFactoryObj.retained;

    var msg = new Messaging.Message(payload);
    msg.destinationName = topic;
    msg.qos = qos;
    msg.retained = retained;
    _client.send(msg);
  }

  var subscribe = function(subFactoryObj) {
    var topic = subFactoryObj.topic;
    var qos = subFactoryObj.qos;
    var onSuccess = subFactoryObj.onSuccess;

    _client.subscribe(topic, {
      qos: qos,
      onSuccess: onSuccess
    });
  }

  /*
  */
  return {
    publish: publish,
    subscribe: subscribe,
  }

})(window);
```
10.2.2 PickMeUp Messenger: The Driver and Passenger location flow

During the approaching phase of the PickMeUp trip, the driver and passenger share location in real time through MQTT subscriptions and publications. The driver application publishes a new location message each time that the HTML5 Geolocation API returns a new location, or when the driver drags the icon on the map. When the driver application receives a new passenger location message, the position of the map icon is updated. Example 10-6 shows the driver application subscribing to the passenger location topic. Example 10-7 shows the driver application publishing a driver location message after the driver's location changes. Example 9-4 shows Messenger processing a new passenger location message and calling the appropriate driver application function.

*Example 10-6  Subscribing to the Passenger Location topic*

```javascript
var subscriptionObj = Messenger.SubscriptionFactory.getPassengerLocationSubscription();
Messenger.subscribe(subscriptionObj);
```

*Example 10-7  Publishing a Driver Location message*

```javascript
var driverGeo = { lon: <longitude>, lat: <latitude> };
var messageObj = Messenger.MessageFactory.getDriverLocationMessage(driverGeo);
Messenger.publish(messageObj);
```
Download, deploy, and run PickMeUp in iOS, Android, and HTML environments

This chapter provides instructions for setting up and running the PickMeUp passenger application (referred to as passenger app in this chapter) provided with this book.

This chapter includes the following topics:

- 11.1, “Set up a PickMeUp iOS project” on page 224
- 11.2, “Set up a PickMeUp Android project” on page 230
- 11.3, “Set up the PickMeUp back end” on page 231
- 11.4, “Set up the PickMeUp HTML5 project” on page 233
11.1 Set up a PickMeUp iOS project

This section describes how to set up your environment to run the PickMeUp iOS passenger app.

11.1.1 Prerequisites

The following prerequisites must be met to run the PickMeUp iOS passenger app:

- You must be using a Macintosh computer to run the PickMeUp iOS passenger app. If you are not using a Macintosh computer, you can still view the source code for the project, if you want.
- Xcode must be installed to open the PickMeUp.xcodeproj file.
- You are required to have an iOS developer certificate to deploy the application to a physical device. Otherwise, the application can only be run on the iOS simulator. With an iOS developer certificate, the application can be installed onto an iOS device, such as an iPad or iPhone.
- Optionally, you must have Git source code management (SCM) installed to the source using the `git clone` command. For details, see 11.1.2, “Obtain the PickMeUp iOS source code” on page 224.

11.1.2 Obtain the PickMeUp iOS source code

The project source is available from GitHub and also from the IBM Redbooks FTP site.

**Obtain the source code from the GitHub repository**

To obtain the source code from the GitHub repository, follow these steps:

1. Open a terminal window.

**Obtain the source code from the Redbooks repository using FTP**

To obtain the source code from the Redbooks repository using FTP, follow these steps:

1. Download the `PickMeUp-iOS.zip` file from the Redbooks FTP site to your computer. For the download instructions, see Appendix B, “Additional material” on page 245.
2. Open a Finder window and navigate to the directory that contains the downloaded file. Figure 11-1 on page 225 shows the `PickMeUp-iOS.zip` file in the PickMeUpWorkspace.
Chapter 11. Download, deploy, and run PickMeUp in iOS, Android, and HTML environments

3. Extract the `PickMeUp-iOS.zip` file.

Figure 11-2 shows the project directory after it is extracted.

11.1.3 Open the PickMeUp iOS project

The following steps show how to open the PickMeUp iOS Xcode project:

1. Open the project directory, `PickMeUp-iOS`.

Figure 11-3 on page 226 shows the contents of the project directory.

---

1 All images on this page were captured using Xcode, which is a trademark of Apple Inc., registered in the U.S. and other countries.
2. Open the **PickMeUp.xcodeproj** file. Figure 11-4 shows the Xcode project after it is opened.

---

2 All images on this page were captured using Xcode, which is a trademark of Apple Inc., registered in the U.S. and other countries.
11.1.4 Configure the project build settings

After opening the project, check the build settings to ensure that they are configured correctly to successfully build and run the application. The following properties need to be set up:

1. Set the Build For Active Architecture only property. To change this setting, click the current value for a list of available options.

   Figure 11-5 shows the property after it is set to the correct value, No.

   ![Figure 11-5](image)

   Figure 11-5 Set the Build Active Architecture Only property to No

2. Set the Install Owner property. To change this setting, click the current value and edit it.

   Figure 11-6 shows the property after it is set to the correct value, $(USER)$.

   ![Figure 11-6](image)

   Figure 11-6 Set the Install Owner property to $(USER)$

---

3 All images on this page were captured using Xcode, which is a trademark of Apple Inc., registered in the U.S. and other countries.
3. Set the Library Search Paths property. To add or change any of the paths listed, click the current value and enter your change. We suggest setting the following paths to non-recursive:
   - $(inherited)
   - $(PROJECT_DIR)
   - $(PROJECT_DIR)/PickMeUp
   - $(PROJECT_DIR)/PickMeUp/Lib

   Figure 11-7 shows the Library Search Paths after they are set to the correct values.4

![Figure 11-7 Set the Library Search Paths property as shown](image)

4. Set up the Code Signing identity and provisioning profile. To view or change this profile, select your developer certificate and provisioning profile, and then set the properties to the following suggested values:

   Code signing identity: <Multiple values>
   - Debug: Don't Code Sign
     - Any iOS SDK: iOS Developer
   - Release: Don't Code Sign
     - Any iOS SDK: iOS Developer

   Provisioning Profile: None

   **Note:** The code signing identity and provisioning profile require a valid developer certificate.

   Figure 11-8 on page 229 shows the Code Signing properties to set.

---

4 All images on this page were captured using Xcode, which is a trademark of Apple Inc., registered in the U.S. and other countries.
11.1.5 Run the application

Follow these steps to run the application:

1. Select the device on which to run the application.

   Click **PickMeUp** (see Figure 11-9, red rectangle) to view a list of devices that the application can be run on. The list contains any devices that are currently connected and any iOS simulators that are available. If you have a valid iOS developer certificate and a device that is provisioned for development, select the iOS device from the menu. Otherwise, pick one of the available simulators.
2. Launch the application.
   To launch the application, click the play (triangle) icon in the upper-left corner of the window (red rectangle in Figure 11-9 on page 229).

3. Optional: Stop the application.
   To stop the application, click the Stop (square) icon in the upper-left corner of the window (red rectangle in Figure 11-9 on page 229).

11.1.6 Run PickMeUp for your iOS project

Instructions are provided to set up and run the sample applications provided with this book.

11.2 Set up a PickMeUp Android project

How to set up your environment to build and run the PickMeUp Android application is described.

11.2.1 Prerequisites

The following prerequisites must be met to run the PickMeUp Android application:

► An Android device with Android 4.1 Jelly Bean (API level 16) or higher is required to run the application.

Notes:

► Running PickMeUp on an emulator is not advised, because PickMeUp is using Google Play Services and Location Services. Using an emulator might cause the application to run with reduced functionality.

► The application was tested using Google Nexus 4, Google Nexus 7, and the Samsung Galaxy S4. For best viewing, we suggest using the screen sizes provided with these devices as the minimum screen sizes.

► A Google account and Google Maps Android API v2 key are required to build the application.

► Android Software Development Kit (SDK) is required to build the PickMeUp application. Stand-alone SDK tools and Eclipse or Android Studio bundle can also be used.

► The PickMeUp source code includes a Gradle build file. This file can be run using Gradle command-line tools, Android Studio integrated development environment (IDE), or Eclipse (with the Gradle plug-in).
11.2.2 Register with Google Maps API

Open a Google account and register for Google Maps API. Any Google account can be used to obtain the key for debugging and development purposes.

**Note:** Before building the PickMeUp application, a Google Maps API key is required. To obtain a Google Maps API key, follow this link. After signing in, press Create:


To add the credentials to an existing key, use the following line:


After the key is generated, replace the google_maps_key entry with the key inside the following file:

<workspace>/PickMeUp/app/src/debug/res/values/google_maps_api.xml

The key starts with the term, Alza.

11.2.3 Android SDK Packages

The following SDK packages are required for building the PickMeUp application:

- Android 4.4.2 (API 19), SDK Platform
- Google Repository
- Android Support Repository
- Android SDK Build-tools 19.1

11.2.4 Run PickMeUp for your Android project

Instructions are provided to set up and run the sample applications described in this book.

11.3 Set up the PickMeUp back end

How to set up and run the PickMeUp back-end application are described. You must already have set up the IBM MessageSight server that you want to use with the PickMeUp application.

11.3.1 Prerequisites

The PickMeUp back end is a Node.js application, so you have to have Node.js installed in your environment. If you want to install it on your local environment, access the Node.js website for more details about setting it up:

http://nodejs.org/

For deployment simplification, you can use the IBM Bluemix environment, which is a Platform as a Service (PaaS) solution from IBM. For more information, see this website:

Use the CloudFoundry command-line interface (CLI) to interact with Bluemix. The CloudFoundry CLI can be downloaded using this link:

https://github.com/cloudfoundry/cli/releases/tag/v6.1.0

### 11.3.2 Register with Bluemix

Sign up for a Bluemix trial at no-charge at this website:

https://ace.ng.Bluemix.net/

**Note:** The PickMeUp back end is lightweight, so the Bluemix trial no-charge option is usually sufficient for trying PickMeUp.

### 11.3.3 Download, deploy, and run PickMeUp

Use the following steps to download, deploy, and run PickMeUp on Bluemix, using CloudFoundry CLI:

1. Download the PickMeUp back-end source code and extract it to your machine.

2. Open the PickMeUp.js file in the package, and edit the configuration code to match the configuration of your MQTT broker (your MessageSight server configuration). A sample is shown in Example 11-1.

   **Example 11-1  MessageSight server configuration for the PickMeUp back end**

   ```javascript
   mqtt: {
       host: '<your.server.ip.goes.here>',
       port: <your.server.port.goes.here>
   }
   ```

3. Target the Bluemix environment for which you registered. See Example 11-2.

   **Example 11-2  Target the Bluemix environment**

   ```bash
   $cf api https://api.ng.bluemix.net/
   ```

4. Log in with your credentials. See Example 11-3.

   **Example 11-3  Log in to Bluemix**

   ```bash
   $cf login -u <username> -p <password>
   ```

5. Target a specific organization and space. See Example 11-4.

   **Example 11-4  Target a Bluemix organization and space**

   ```bash
   $cf target -o <orgName> -s <spaceName>
   ```

6. Navigate to the directory to which you extracted the PickMeUp back-end code.

7. Deploy the code using the $cf push Pickmeup --no-route command shown in Example 11-5.

   **Example 11-5  Deploy the PickMeUp code**

   ```bash
   $cf push Pickmeup --no-route
   ```
8. Optional: After the application is deployed, you can view the log file when you are running PickMeUp. Use the $cf logs Pickmeup command shown in Example 11-6.

Example 11-6  Tracing the logs of the application

$cf logs Pickmeup

11.4 Set up the PickMeUp HTML5 project

The HTML5 driver application (referred to as the driver app) consists of HTML, JavaScript, and Cascading Style Sheets (CSS) files. The application can be run using the following steps:

1. Extract the PickMeUp HTML5 source code to your machine.
2. Optional: Modify <driver_app>/js/Messenger.js to replace the MQTT server and port configuration with that of your MessageSight server.
3. Deploy the HTML5 assets using a web server.
4. In a websockets-compatible, modern browser (examples are provided in 10.1, “Advantages of developing an HTML5 PickMeUp application” on page 218), open index.html at the address specified by your web server configuration.
The MQTT protocol

The MQTT protocol is a lightweight network protocol used for publish and subscribe messaging. This protocol is optimized for wireless remote devices that might have limited bandwidth and intermittent connectivity. This appendix describes the following concepts of the MQTT protocol:

- “Quality of service (QoS) levels and flow” on page 236
- “QoS determination” on page 238
- “Impact of QoS level on performance” on page 239
- “The MQTT client identifier” on page 239
- “MQTT durable and non-durable subscribers” on page 240
- “MQTT persistence” on page 240
- “The MQTT header” on page 241
- “The MQTT keep alive timer” on page 242
- “Delivery of the MQTT retry message” on page 243
- “The MQTT last will and testament” on page 243
- “The MQTT retained flag on messages” on page 244
- “The TCP/IP stack” on page 244

Note: The IBM developerWorks® website contains the MQ Telemetry Transport (MQTT) V3.1 Protocol Specification at this website:

Quality of service (QoS) levels and flow

Quality of service (QoS) levels determine how each MQTT message is delivered, and they must be specified for every message that is sent through MQTT. It is important to choose the proper QoS value for every message, because this value determines how the client and the server communicate to deliver the message.

A real-life comparison can be made with letters that are mailed using the postal service. An informal letter to a friend can be dropped into a mailbox, and the sender might never think about it again. The sender expects the letter to arrive at its destination, but there are no significant consequences if it does not arrive. In contrast, a letter to which the recipient must respond is more important, so the sender might choose a higher level of delivery service that provides proof that the letter was received at its destination. In each situation, the choices made by the sender of the letter are comparable to choosing QoS levels in MQTT.

QoS Level 0: At most once delivery

Messages that are selected as *At most once message delivery* are delivered according to the best effort of the underlying network. A response is not expected, and no retry semantics are defined in the protocol. The message arrives at the MQTT server either one time or not at all. This is the lowest QoS level. The MQTT client or server attempts to send the message without waiting for a confirmation of receipt. There are no steps taken to ensure that the message is delivered, other than the features provided by the TCP/IP layer. Also, if the message fails to deliver, there is no retry attempt made by the MQTT layer. Therefore, if the client is sending a message, it will arrive at the MQTT server either one time or never. A QoS 0 message can get lost if the client unexpectedly disconnects or if the server fails. From a performance perspective, this adds value because it is the fastest way to send a message using MQTT.

The MQTT command message used is PUBLISH. No other command messages flow for QoS 0 messages. Table A-1 shows the QoS 0 protocol flow.

<table>
<thead>
<tr>
<th>Table A-1</th>
<th>QoS Level 0 protocol flow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td><strong>Message and direction</strong></td>
</tr>
<tr>
<td>QoS = 0</td>
<td>PUBLISH</td>
</tr>
</tbody>
</table>

QoS Level 1: At least once delivery

With QoS set to Level 1, the message is delivered at least one time. The MQTT client or the server attempts to deliver the message at least one time, but there can also be a duplicate message. The receipt of a message by the MQTT server is acknowledged by a PUBACK message. If there is an identified failure of either the communications link or the sending device, or if the acknowledgment message is not received after a specified period of time, the sender resends the message with the DUP bit set in the message header. The message arrives at the MQTT server at least one time. Both SUBSCRIBE and UNSUBSCRIBE messages use QoS 1.
If the client does not receive a PUBACK message (either within a time period defined in the application, or if a failure is detected and the communications session is restarted), the client resends the PUBLISH message with the DUP flag set. When it receives a duplicate message from the client, the MQTT server republishes the message to the subscribers, and sends another PUBACK message. On the client side, the implementation of the MQTT protocol also provides an additional feature known as MQTT persistence. The MQTT persistence layer is not described in the MQTT specification, but it is normally available with MQTT client implementations. When a QoS 1 message is published to the server, the client needs to wait for the acknowledgment to arrive. There can be a program termination or a crash at the client device. When the client is started again, it will need to resume from the point it left before the crash. Therefore, the message is stored in a persistence layer, such as disk, and retrieved soon after the reconnecting back to the MQTT server.

The MQTT command messages used are PUBLISH and PUBACK. When the publish happens, the message will be logged to the MQTT persistence and removed when PUBACK is received. A message with QoS 1 has a Message ID in the message header. Table A-2 shows the QoS 1 protocol flow.

### Table A-2  QoS Level 1 protocol flow

<table>
<thead>
<tr>
<th>Client</th>
<th>Message and direction</th>
<th>MQTT server</th>
</tr>
</thead>
<tbody>
<tr>
<td>QoS = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DUP = 0</td>
<td>PUBLISH</td>
<td>Actions:</td>
</tr>
<tr>
<td>Message ID= x</td>
<td>----------------------</td>
<td>-- Store message in database</td>
</tr>
<tr>
<td>Action: Discard message</td>
<td>PUBACK</td>
<td>-- Publish message to subscribers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### QoS Level 2: Exactly once delivery

This is the highest level of QoS. Additional protocol flows that are higher than QoS 1 ensure that duplicate messages are not delivered to the receiving application. When QoS 2 is used, the message is delivered one time only. The MQTT client or the server ensures that the message is sent only one time. This QoS must be used only when duplicate messages are not wanted. From a performance perspective, there is a price to be paid in terms of network traffic and processing power.

The MQTT command messages used are PUBLISH, PUBREC, PUBREL, and PUBCOMP. The message is sent in the PUBLISH flow, and the client stores that message in the MQTT persistence layer, if used. The message remains locked on the server. PUBREC is sent by the server in response to PUBLISH. PUBREL is dispatched to the server from the client in response to PUBREC. After PUBREL is received by the MQTT server, the messages can be dispatched to the subscribers and PUBCOMP is sent back in response to the PUBREL. A QoS 2 message has a Message ID in the message header.

If a failure is detected, or after a defined time period, each part of the protocol flow is retried with the DUP bit set. The additional protocol flows ensure that the message is delivered to subscribers one time only.

Because QoS 1 and QoS 2 indicate that messages must be delivered, the MQTT server stores messages in a database. If the MQTT server has problems accessing this data, messages might be lost. Table A-3 on page 238 shows the QoS 2 protocol flow.
Assumptions of QoS levels 1 and 2

In any network, it is possible for devices or communication links to fail. If this happens, one end of the link might not know what is happening at the other end. This situation is known as *in doubt windows*. In these situations, assumptions have to be made about the reliability of the devices and networks involved in message delivery.

From the MQTT perspective, it is assumed that the client and server are generally reliable, and that the communications channel is more likely to be unreliable. If the client device fails, it is typically a catastrophic failure, rather than a transient one. The possibility of recovering data from the device is low. Certain devices have nonvolatile storage, for example, flash ROM. The provision of more persistent storage on the client device protects the most critical data from certain modes of failure.

Beyond the basic failure of the communications link, the failure mode matrix becomes complex, resulting in more possible scenarios than the specification for MQTT can handle.

The time delay (the retry interval) before resending a message that has not been acknowledged is specific to the application, and it is not defined by the protocol specification.

QoS determination

When the client subscribes to a topic, the QoS level is selected. Consider a scenario where Publisher A is sending messages to a topic at QoS 2. A subscriber can subscribe at QoS 0, so that the messages to the client are delivered with QoS 0. The QoS value is further used in SUBSCRIBE and UNSUBSCRIBE requests. A subscribe request from the client is a QoS 1 request, to which the server responds with SUBACK, which ensures that the subscription has occurred.
Impact of QoS level on performance

There is a simple rule when considering the performance impact of QoS: *The higher the QoS, the lower the performance*. Let us evaluate performance corresponding with higher QoS. Suppose the time taken for sending a PUBLISH message is $pt$. If QoS 0 is used, the total time taken to transfer $n$ number of messages is $npt$. Now, for QoS 1, the PUBACK message (that is, the reply to the PUBLISH message) flows from the server to the client. This is a 2-byte message and typically takes less time than $pt$ takes. Therefore, the time for the reply to the PUBLISH message is called $mt$. So, the time taken for transferring $n$ messages is $n(pt + mt)$.

For example, for QoS 2, the PUBREC, PUBREL, and PUBCOMP messages are flowing. Therefore, $n$ number of messages takes approximately $n(pt + 3mt)$. So, if 10 messages need to be transferred from client to server, and $pt$ is 1 second and $mt$ is 0.4 seconds, a QoS 0 message takes 10x1 (or 10 seconds). By comparison, QoS 1 messages take 10(1+0.4) (or 14 seconds), and QoS 2 messages take 10(1+1.2) (or 22 seconds).

The MQTT client identifier

The MQTT protocol requires that a client ID is defined by the client, so that the client is uniquely defined on a network. In simple terms, the client specifies a unique string for connecting to the MQTT server. There are several ways of choosing a client ID, for example:

- A sensor installed in a particular location can use the location code as the client ID.
- A mobile device with a network connection can use the Media Access Control (MAC) address or unique device ID as the client ID.

MQTT restricts the client ID length to 23 characters. Certain situations require that the client ID is shortened. As an example, if the MAC address of the client's machine is used as the MQTT client ID, the client needs to shorten the ID to meet the 23-character limit. When shortening the client ID, the client ID cannot be the same as any other client ID on the network. To keep the client ID short and unique, we suggest that the client introduce a reliable ID generation mechanism. For instance, you might create a client identifier from the 48-bit device MAC address. If transmission size is not a critical issue, you might use the remaining 17 bytes to make the address easier to administer, such as human-readable text in the identifier.

If two clients were to have the same client identifier, one of the clients might receive a message and the other client might not. The MQTT server tracks the pending messages that are to be sent to a client based on the client ID. Therefore, if a client has been using QoS 1 or QoS 2, and has subscribed to any topic and disconnected from the server, the server saves the messages that arrived for the client when it was disconnected. After the client reconnects, the server sends those messages to the client. If a second MQTT device uses the same client ID and connects to the server, the server will send the saved messages to the second device.

Another scenario related to the client IDs is duplicate connections. Assume that a particular device using client ID DeviceA is connected to the MQTT server. If another client logs in with the same client ID DeviceA, the first (older) client must be disconnected by the server before completing the CONNECT flow of the second (new) client. This is an optional feature of an MQTT server.
MQTT durable and non-durable subscribers

A **durable subscriber** in any client can receive all the messages published on a topic, including messages published when the client was inactive. A client is said to be inactive in the following cases:

- A client has connected and subscribed to a topic and later disconnected with the server without unsubscribing to that topic.
- A client has connected and subscribed to a topic and later is not on the network due to a network issue or a loss of the connection to the server.

A **non-durable subscriber** is any client that does not intend to receive messages published when it is inactive. A client is said to be non-durable in the following cases:

- A client always uses the clean session flag set to `true` when it connects to the server.
- A client always unsubscribes for all the topics it is subscribed to, before disconnecting.

In MQTT, QoS 0 messages are never persisted when a durable subscriber is inactive. Only QoS 1 and QoS 2 messages are persisted by the server and sent when the client becomes active. MQTT further provides a facility that allows MQTT clients to refrain from receiving messages when they were disconnected by setting the clean start flag to `true`. The following example flow is for a typical durable MQTT subscriber:

1. MQTT ClientA connects to the server by specifying the clean session flag as `false`.
2. ClientA subscribes to the topic with QoS 2.
3. MQTT ClientB connects to the server and publishes QoS 1 and QoS 2 messages to the topic.
4. ClientA receives the messages.
5. ClientA disconnects from the server. Note that ClientA did not unsubscribe.
6. ClientB publishes more QoS 1 and QoS 2 messages.
7. ClientA connects to the server by specifying the clean session flag as `false`.
8. ClientA receives all of the messages that were published when it was inactive and all future messages that are published to the topic.

Durable subscribers are useful when the subscribing MQTT client needs all of the messages that are published to the topic it subscribes to. For durable subscribers, an MQTT client is assured to receive any messages when it was disconnected or even had lost a connection. Although durable subscriptions is a nice feature, it adds the additional requirement of the server to hold messages until the client reconnects to the server.

**MQTT persistence**

The MQTT protocol is designed with the assumption that the client and server are generally reliable. This means that the client machine and the server machine will not crash, hang, or have power failure issues. The assumption might prove to be costlier in certain situations for the clients and, therefore, a feature known as **MQTT client persistence** is provided. The local persistence store is used to achieve QoS 1 and QoS 2 level message flows. For example, before the client sends the **PUBLISH** command, the client stores the message data on disk or any other available storage.
The client, when it receives an acknowledgment, deletes the message from the disk. In the case of a power failure followed by a restart of the client application, one of the first actions after the reconnect is to check for pending client messages and to send them. On the server side, this feature is generally managed by the messaging engine itself.

**Message delivery from JMS to MQTT**

When messages are sent from a Java Message Service (JMS) application to MQTT applications, the JMS reliability and persistence map to MQTT QoS levels. For the fastest but least reliable messaging between JMS and MQTT, nonpersistent JMS messages can be sent with the IBM MessageSight `DisableACK` property set to `true`. This configuration provides QoS 0 for MQTT subscriber applications.

For the best reliability but slowest messaging rates, send persistent JMS messages and user per-message acknowledgment. This configuration provides up to QoS 2 for MQTT subscribers. Table A-4 shows the possible combinations for message delivery from JMS to MQTT.

<table>
<thead>
<tr>
<th>JMS message type sent</th>
<th>QoS of matching MQTT subscription</th>
<th>Reliability of delivered message</th>
<th>Persistence of delivered message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Either nonpersistent or persistent</td>
<td>QoS 0</td>
<td>At most one time (QoS 0)</td>
<td>Nonpersistent</td>
</tr>
<tr>
<td>Nonpersistent, acknowledgments turned off</td>
<td>QoS 1</td>
<td>At most one time (QoS 0)</td>
<td>Nonpersistent</td>
</tr>
<tr>
<td>Nonpersistent</td>
<td>QoS 1</td>
<td>At least one time (QoS 1)</td>
<td>Nonpersistent</td>
</tr>
<tr>
<td>Persistent</td>
<td>QoS 1</td>
<td>At least one time (QoS 1)</td>
<td>Persistent</td>
</tr>
<tr>
<td>Nonpersistent, acknowledgments turned off</td>
<td>QoS 2</td>
<td>At most one time (QoS 0)</td>
<td>Nonpersistent</td>
</tr>
<tr>
<td>Nonpersistent</td>
<td>QoS 2</td>
<td>At least one time (QoS 1)</td>
<td>Nonpersistent</td>
</tr>
<tr>
<td>Persistent</td>
<td>QoS 2</td>
<td>Exactly one time (QoS 2)</td>
<td>Persistent</td>
</tr>
</tbody>
</table>

**The MQTT header**

The MQTT protocol only requires a 2-byte header. To understand the overall payload size, consider a simple scenario in which a client sends a message that reads “Hello” 10 times. Additionally, the client indicates to the server that the destination is topic `a/b` for every message. We can calculate how many additional bytes of data will flow from the client to the server for one of the scenarios.
For sending the messages using MQTT, the computation of total number of bytes is shown:

- **CONNECT**: Fixed (2 bytes) + Variable (12 bytes) = 14 bytes
- **CONNACK**: Fixed (2 bytes) + Variable (2 bytes) = 4 bytes
- **PUBLISH**: Fixed (2 bytes) + Variable: Length (a/b) = 3

So, two bytes for representing the topic name, followed by the topic name, followed by the message ID, followed by the payload, is 2 + 3 + 2 + 5 = 12 bytes.

- There are 10 publications happening... therefore (12) 10 = 120 bytes
- **DISCONNECT**: Fixed (2 bytes) = 2 bytes

Therefore, this scenario needs 14 + 4 + 120 + 2 = 140 bytes.

As you can see from this example, the overhead for every message is fixed at 2 bytes. The total overhead is 140 - 80 = 60 bytes for this scenario.

### The MQTT keep alive timer

An MQTT server is able to determine whether the MQTT client is still in the network (and vice versa) by using the keep alive timer. The TCP/IP timeout and error handling mechanisms are at the network layer, but the MQTT client and server do not need to depend on that. The client says hello to the server, and the server responds back, acknowledging it. That is how simple the keep alive timer is.

For MQTT, the client only needs to tell the server that it is alive when no interaction has occurred between them for a certain period of time. The time period is a configurable option. This option can be set when connecting to the server by the client. It is the client that chooses the keep alive time; the server just keeps a record of the value in the client information table on the server side.

There are two messages that constitute the keep alive interaction:

- **PINGREQ**: The ping request is sent by the client to the server when the keep alive timer expires.
- **PINGRESP**: The ping response is the reply sent by the server to a ping request from the client.

Both of these messages are short, 2-byte messages. They are not associated with any QoS. The PINGREQ is sent only one time, and the client waits for the PINGRESP.

Internally, there is a simple timer that triggers in the client. The timer determines whether there has been recent activity between the client and the server. If not, a PINGREQ is sent. The client waits until a PINGRESP is received. On the server side, if there is no message received from the client within the keep alive time period, the server disconnects the client. However, the server offers a grace period to the client, which is an additional 50% of the keep alive time period.

When choosing a keep alive time for a client, there are certain considerations. If the MQTT client is less active, sending, for example, one or two messages within an hour, and if the keep alive timer is set to a low value, for example, 10 seconds, the network might be flooded with PINGREQ and PINGRESP messages.
Alternatively, if the keep alive timer is set to a high value, for example, one hour and the client goes out of the network, the server will not know that the client has gone away for a long time. This can affect administrators who monitor the connected clients. Also, the server will keep retrying PUBLISH messages. Set the keep alive timer to suit your specific needs, based on the message flow and the amount of time that the client can be idle.

**Delivery of the MQTT retry message**

Consider a scenario where a QoS 1 message is published from a client to a server, but the PUBACK is not received from the server. The MQTT client will retry sending the message after a specified retry timeout period. The retry timeout period is not the same as the keep alive timer. The retry timeout period is the maximum amount of time the client waits for the server to send a response, or vice versa. This retry reoccurs until there is an error from the TCP/IP layer. That is, if there is any type of socket exception, retry processing ceases. In cases where QoS 2 message delivery is not used, duplicate messages might be sent, due to retry by the MQTT client or server. Therefore, when a retry occurs, the message is marked with a duplicate flag. The application receiving the message can identify whether the received message is a duplicate.

It is possible that a duplicate message is received by an MQTT client when using QoS 1, due to retries by MQTT. On slow and fragile networks, you can observe a large number of duplicate messages that have occurred because of retry processing. If the retry timeout period is short, the network can be clogged with a large number of duplicates.

**The MQTT last will and testament**

When a client connects to the server, the client can define a topic and a message that needs to be published automatically if it unexpectedly disconnects. This is called the last will and testament, commonly referred to as the Will message. The concept of the last will and testament is introduced in “Publications at connection time” on page 174.

If an unexpected disconnect occurs when the client does not send a DISCONNECT command to the server, the publication is sent by the telemetry service when it detects that the connection to the client has broken, without the client requesting a disconnect. The client might have experienced a loss in network connection or an abrupt termination of the client program. The publication is sent to a server for publishing messages to another application that is monitoring the status of connected clients.

When a client connects to the server, the following last will and testament parameters can be specified:

- The topic to which the Will message needs to be published
- The message that needs to be published
- The QoS of the message that will be published on the topic
- The retain flag that signifies whether the message will be retained

When the client unexpectedly disconnects, the keep alive timer on the server side detects that the client has not sent any message or a keep alive PINGREQ. When this situation occurs, the server publishes the last will and testament message on the topic specified by the client.
The last will and testament feature can be useful in certain scenarios. For remote MQTT clients, this feature can be used to detect when the device goes out of the network. The last will and testament feature can be used to create notifications for an application that is monitoring client activity.

The MQTT retained flag on messages

MQTT provides a feature in which it holds a message for a topic, even after the message is delivered to the connected subscribers. This feature is achieved by using a retained flag on the publish message.

The publisher of the message sets this flag on the message during publishing. The following example flow shows you how to understand retained messages:

1. Client A connects to the server and subscribes to topics a/b.
2. Client B connects to the server and publishes the message “Hello” with the retain flag to topics a/b.
3. Client A receives the message without the retain flag set to 0.
4. Client C connects to the server and subscribes to topics a/b.
5. Client C receives the message with the retain flag set to 1.

Even if the server is restarted, the retained message will not be lost.

Important: Only one retained message is held per topic.

The retained publications are primarily used to maintain state information. If a particular topic is used to publish a state message from a device, the messages can be retained messages. The advantage is that the new monitoring program that connects can subscribe to this topic and get information about the last published state messages from the device.

The TCP/IP stack

When designing applications for mobile wireless networks, considerations need to be made for TCP/IP. MQTT is like other protocols, including HTTP or FTP, for the TCP/IP stack. TCP/IP is the communication protocol for the Internet. It defines how clients and servers communicate with each other. All of the major server OSs support TCP/IP. Client devices also have operating systems that support TCP/IP. Although MQTT adds only 2 bytes to a message header, more data flows in the network layers. The additional header data cannot be exactly computed on a per MQTT message basis. The IP header, frame header, and other keep alive flows at the network layer add to additional data flows when using MQTT clients.

TCP/IP port 1883 is reserved for use with MQTT. TCP/IP port 8883 is also registered for using MQTT over Secure Sockets Layer (SSL).
Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the web material

This book is available in softcopy form on the Internet from the IBM Redbooks web server. Point your web browser at the IBM Redbooks FTP site:
ftp://www.redbooks.ibm.com/redbooks/SG248228
Alternatively, you can go to the IBM Redbooks website:
ibm.com/redbooks
Select Additional materials and open the directory that corresponds with the IBM Redbooks form number, SG248228.

Using the web material

The additional web material that accompanies this book includes the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PickMeUp-iOS.zip</td>
<td>PickMeUp is a ride sharing application powered by MQTT. Passengers have the option of using various native mobile applications for interacting with the service. This is the native iOS version of the mobile passenger application.</td>
</tr>
<tr>
<td>Chapter2app.zip</td>
<td>This is the HTML and JavaScript implementation for the ITSO Tennis News application referenced in Chapter 2, “Getting started with MQTT” on page 21.</td>
</tr>
</tbody>
</table>
System requirements for downloading the web material

The web material requires the following system configuration:

**Platform:** The PickMeUp application code (PickMeUp-iOS.zip) that you can download from the IBM Redbooks FTP site runs on iOS based mobile devices. The book also shows how to create Android and HTML based versions of this application.

The ITSO Tennis News application (Chapter2app.zip) runs on the Eclipse Paho platform. For the current prerequisites, see this website:

http://www.eclipse.org/paho/clients/java/

Downloading and extracting the web material

Create a subdirectory (folder) on your workstation, and extract the contents of the web material compressed file into this folder.
Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- *Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry*, SG24-8054

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- A demo collection that showcases dynamic HTML5 applications using IBM MessageSight and MQTT for real-time analytics, communication, and collaboration:
  http://m2m.demos.ibm.com

- Restoring the MessageSight configuration data to the same appliance:

- Restoring the MessageSight configuration data to a different appliance:

- Message hub commands in the MessageSight IBM Knowledge Center:

- A trial version of the key fob remote application is available at the following URL:

- Message hub commands in the MessageSight IBM Knowledge Center:
- **Configuring an external LDAP server in the IBM MessageSight Knowledge Center:**

- **MQ Telemetry Transport (MQTT) V3.1 Protocol Specification** at this website:

- Two online video tutorials related to the key fob application are also available. These describe the MQTT application:
  - **IBM MessageSight: Powering an MQTT application**, which is accessible at this website:
    http://bit.ly/keyfobvideo1
  - **IBM MessageSight: Authenticating an MQTT application**, which is accessible at this website:

- The Paho project:
  http://www.eclipse.org/paho/

- IoT.eclipse.org is making the Internet of Things (IoT) development simpler:
  http://iot.eclipse.org/

- IBM MessageSight v1.0.0.1:
  http://ibm.co/1qzajLo

- An evaluation version of WebSphere MQ is available at this website:

- MQTT messaging protocol:
  http://www.mqtt.org

- RSMB:

- For more information about Mosquitto or to download the Mosquitto broker, see this website:
  http://mosquitto.org/

- The Eclipse Paho MQTT clients can be downloaded from this website:
  http://www.eclipse.org/paho/download.php

- IBM Mobile Messaging and machine-to-machine (M2M) Client Pack MA9B:
  http://www-01.ibm.com/support/docview.wss?uid=swg27038199

- Apache Maven:
  http://maven.apache.org/download.cgi

- MessageSight Ethernet interfaces configuration guide available in the MessageSight IBM Knowledge Center:
  http://www-01.ibm.com/support/knowledgecenter/SSCGGQ_1.0.0/com.ibm.ism.doc/Administering/ad00212_.html
- MessageSight V1.1 Command Reference guide in the MessageSight IBM Knowledge Center website:
  

- Message hub commands in the MessageSight IBM Knowledge Center:
  

- FIPS:
  
  http://www.itl.nist.gov/fipspubs/

- Configuring an external LDAP server in the IBM MessageSight Knowledge Center:
  

- MQTT V3.1 Java and C clients, which IBM contributed, and which you can download from the following web pages:
  
  http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.java.git/
  http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.c.git/

- Node.js website:
  
  http://nodejs.org/

- IBM Bluemix environment (Platform as a Service (PaaS) solution from IBM). For more information, see this website:
  

- Bluemix trial at no-charge at this website:
  
  https://ace.ng.Bluemix.net/

- CloudFoundry CLI:
  
  https://github.com/cloudfoundry/cli/releases/tag/v6.1.0

- All the scoreboards are sponsored and developed by IBM using IBM MessageSight at both the US Open and Wimbledon:
  
  - US Open:
    
  
  - Wimbledon:
    
    http://www.wimbledon.com/en_GB/scores/

  - The Australian open live score application can be accessed from the following URL:
    
    http://www.ausopen.com/en_AU/scores/

---

**Help from IBM**

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services
Building Real-time Mobile Solutions with MQTT and IBM MessageSight
Building Real-time Mobile Solutions with MQTT and IBM MessageSight

Provides practical guidance to getting started quickly with MQTT and IBM MessageSight

Builds a mobile application (PickMeUp) by using MQTT and IBM MessageSight

Shows typical usage patterns and guidance to expand the solution

MQTT is a messaging protocol designed for the Internet of Things (IoT). It is lightweight enough to be supported by the smallest devices, yet robust enough to ensure that important messages get to their destinations every time. With MQTT devices, such as energy meters, cars, trains, mobile phones and tablets, and personal health care devices, devices can communicate with each other and with other systems or applications.

IBM MessageSight is a messaging appliance designed to handle the scale and security of a robust IoT solution. MessageSight allows you to easily secure connections, configure policies for messaging, and scale to up to a million concurrently connected devices.

This IBM Redbooks publication introduces MQTT and MessageSight through a simple key fob remote MQTT application. It then dives into the architecture and development of a robust, cross-platform Ride Share and Taxi solution (PickMeUp) with real-time voice, GPS location sharing, and chat among a variety of mobile platforms. The publication also includes an addendum describing use cases in a variety of other domains, with sample messaging topology and suggestions for design.

For more information: ibm.com/redbooks