



MQTT – Message Queue Telemetry Transport Protocol with .NET Core



Jawad Hasan

20 Oct 2020 CPOL

How to use MQTT Protocol in your .NET Core Applications

Publisher/Subscriber Pattern with “a small code footprint and on-the-wire footprint” message protocol

Introduction

Pub/Sub pattern is a common requirement for decoupled softwares. There are various technologies and protocols available. In this post, I will show you how to implement in your .NET Core applications using MQTT protocol.

What is MQTT

MQTT is a message protocol with “a small code footprint and on-the-wire footprint”. MQTT is a publish-subscribe-based messaging protocol which is built on top of TCP/IP.

The protocol uses a **publish/subscribe** architecture in contrast to HTTP with its request/response paradigm. Publish/Subscribe is event-driven and enables messages to be pushed to clients.

The central communication point is the **MQTT broker**, it is in charge of dispatching all messages between the **senders** and the rightful **receivers**.

Each client that **publishes** a message to the broker, includes a **topic** into the message. **The topic is the routing information for the broker.**

Each client that wants to receive messages **subscribes** to a certain topic and the broker delivers all messages with the matching topic to the client.

Therefore, the **clients don't have to know each other**, they only communicate over the topic.

This architecture enables highly scalable solutions without dependencies between the data **producers** and the data **consumers**.

... and What is with REST?

- HTTP/REST is useful to handle documents and resources.
- MQTT is useful to handle messages.
- HTTP/REST can be complex and is not always the best solution for simple messages.
- The MQTT packet size is 2 byte + payload.
- MQTT supports 1-to-1, 1-to-many and many-to-many messages.
- Request and response **vs** publisher and subscriber.

Architecture

The difference to HTTP is that a client doesn't have to pull the information it needs, but the broker pushes the information to the client, in case there is something new.

Therefore each MQTT client has a **permanently open TCP connection** to the broker. If this connection is interrupted by any circumstances, the MQTT broker can buffer all messages and send them to the client when it is back online.

As mentioned before, the central concept in MQTT to dispatch messages are topics. **A topic is a simple string that can have more hierarchy levels, which are separated by a slash.**

A sample topic for sending temperature data of the living room could be ***house/living-room/temperature***.

On the one hand, the client can subscribe to the exact topic or on the other hand use a wildcard. The subscription to ***house/+/temperature*** would result in all messages sent to the previously mentioned topic *house/living-room/temperature* as well as any topic with an arbitrary value in the place of living room, for example, *house/kitchen/temperature*.

The plus sign is a **single level wild card** and only allows arbitrary values for one hierarchy. If you need to subscribe to more than one level, for example to the entire subtree, there is also a **multilevel wildcard** (#). It allows to subscribe to all underlying hierarchy levels. For example, *house/#* is subscribing to all topics beginning with *house*.

Payload

- MQTT is payload agnostic. You can use any of the following:
 - a simple byte array
 - a simple string
 - or a JSON

PUBLISH to **home/livingroom/light/1** message

Security

- SSL/TLS support
- Username/Password
- Encrypt payload (data/payload agnostic)
- IOT security should not be underestimated!
- SSL/TLS is a must-have

Code Sample

I have built a sample .NET Core console application to test the library. Following are code screenshots, which are self explanatory and you can download the code from git if needed. Let me know in the comments if something is not clear.

The solution contains three projects as follows and all of the projects have a reference to MQTTnet. One application act as a publisher, the other as subscriber and the third one as a broker to illustrate the main blocks of messaging system.

Broker

Both, Publisher and Subscriber connects to Broker.

Publisher

Here is the code for **SimulatePublish** method:

Subscriber

Execution

Following is screenshot of running the solution:

>> Publisher and Subscribers are connected with the Broker.

and here you can see publisher/subscriber and broker in-action:

Summary

This was a very basic introduction to MQTT and its usage. To keep the discussion simple, I kept the code to very minimal. Publisher/Subscriber pattern is very powerful and allows us to create decoupled application easily and use of MQTTNet library makes it very easy to implement this pattern in our applications. You can download the sample from my git repository on the source code link below. Also I will suggest to check the links in reference for more insight. Till next time, happy coding.

Git Repository Link

- <https://github.com/jawadhasan/mqttBasic.git>

References

- <https://www.hivemq.com/blog/how-to-get-started-with-mqtt/>
- <https://github.com/chkr1011/MQTTnet/wiki/Client>

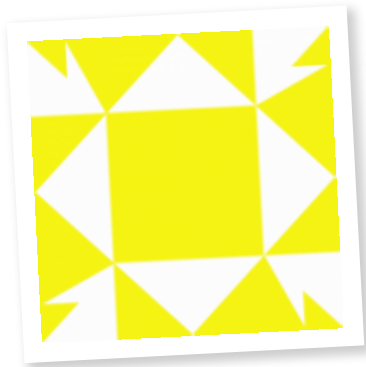
History

- 19th October, 2020: Initial version


License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOl\)](#)

About the Author



JawadHasan

Germany 

Hi there! I'm Jawad and I'm a software solutions engineer and this is my website. I live in Dusseldorf, Germany, have a great interest in books and movies, and Astrophysics as well.

<https://hexquote.com/aboutme/>

Comments and Discussions

 **3 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/5283088/MQTT-Message-Queue-Telemetry-Transport-Protocol-wi> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Article Copyright 2020 by JawadHasan
Everything else Copyright © [CodeProject](#),
1999-2020

Web02 2.8.20201015.3