Best MQTT Documentation

Offline Version

Last generated: January 04, 2022

© 2022 Copyright 2021 Tivadar György Nagy. All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Table of Contents

Introduction

Summary: Best MQTT is an MQTT v5 protocol implementation that works on all major platforms, including WebGL.

Introduction

MQTT is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth. MQTT today is used in a wide variety of industries, such as automotive, manufacturing, telecommunications, oil and gas, etc.

Best MQTT built on top of Best HTTP/2 to use the best available solution under Unity3D. By using Best HTTP/2 it can target the same platforms, including WebGL and use it for TCP and WebSocket transports, reduce memory usage with BufferPools, secure connecctions with TLS and use its logging infrastructure.

Supported Platforms

Same platforms are supported as Best HTTP/2 (page 0). Under WebGL only the WebSocket transport can be used and certificate based authentication doesn't work.

Contact, Support

For support, feature requests or general questions you can email me at besthttp@gmail.com or join to the plugin's discord server .

Client Setup

Summary: A step-by-step guide to set up and use MQTTClient.

Installation

Import the Best MQTT package and all of its dependencies through the Unity Editor's Package Manager window. If all packages are imported properly, no further steps are required.

Initial setup

After successfully imported both the Best MQTT and Best HTTP/2 packages, you can create a new C# Script file in your Unity project and add the plugin's namespace ' `BestMQTT` ' and ' `BestMQTT.Packets.Builders` ' somewhere around the other usings and delete the Update function as we will not going to use it:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using BestMQTT;
using BestMQTT.Packets.Builders;

public class MQTT : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }
}
```

Create a new scene in Unity and assign this script to a gameobject.

Creating a ConnectionOptions instance

First we have to create a `ConnectionOptions` instance to pass it to the `MQTTClient` 's constructor in the next step. `ConnectionOptions` contains connection related information like the `Host` and `Port` properties. The easiest way to create a `ConnectionOptions` is using `ConnectionOptionsBuilder` :

```
var options = new ConnectionOptionsBuilder()
    .WithTCP("broker.emqx.io", 1883)
    .Build();
```

The `WithTCP("broker.emqx.io", 1883)` line tells the plugin to try to connect to the “broker.emqx.io” host with port 1883 using the TCP transport. If we want to use TLS to encrypt the server-client communication add the .WithTLS() call:

```
var options = new ConnectionOptionsBuilder()
    .WithTCP("broker.emqx.io", 8883)
    .Build();
```

In this guide i used EMQ X’s public broker , but there are many others to choose from .

> The server must support TLS v1.1 or v1.2 and hosted on a different port!

To use Websocket as the transport protocol instead of `WithTCP` the `WithWebSocket` function must be used.

> The TCP transport isn’t available under WebGL!

Creating the MQTTClient

With the newly created ConnectionOptions instance we can create the `MQTTClient` instance:

```
var client = new MQTTClient(options);
```

Add general events

The client going to start to connect to the server when instructed so with BeginConnect `or` ConnectAsync`. We can freely add and modify the client until one of these are called without fearing that any event is missed. Let’s add a few event handlers to catch client related events:

```
client.OnStateChanged += OnStateChanged;
client.OnDisconnect += OnDisconnected;
client.OnError += OnError;
```

And add the implementation of the event handlers:

```
// Called when the MQTTClient transfered to a new internal state.
private void OnStateChanged(MQTTClient client, ClientStates oldState, ClientStates newState)
{
    Debug.Log($"{oldState} => {newState}");
}

// Called when the client disconnects from the server. The disconnection can be client or serve
r initiated or because of an error.
private void OnDisconnected(MQTTClient client, DisconnectReasonCodes code, string reason)
{
    Debug.Log($"OnDisconnected - code: {code}, reason: '{reason}'");
}

// Called when an error happens that the plugin can't recover from. After this event an OnDiscon
nected event is raised too.
private void OnError(MQTTClient client, string reason)
{
    Debug.Log($"OnError reason: '{reason}'");
}
```

These events are not tightly related to the MQTT protocol, but they can give a good understanding when and what happens with the client connection.

A more compact way to create and setup `ClientOptions` and the client itself is to use `MQTTClientBuilder`:

```
client = new MQTTClientBuilder()
                .WithOptions(new ConnectionOptionsBuilder().WithTCP("broker.emqx.io", 1883))
                .WithEventHandler(OnStateChanged)
                .WithEventHandler(OnDisconnected)
                .WithEventHandler(OnError)
                .CreateClient();
```

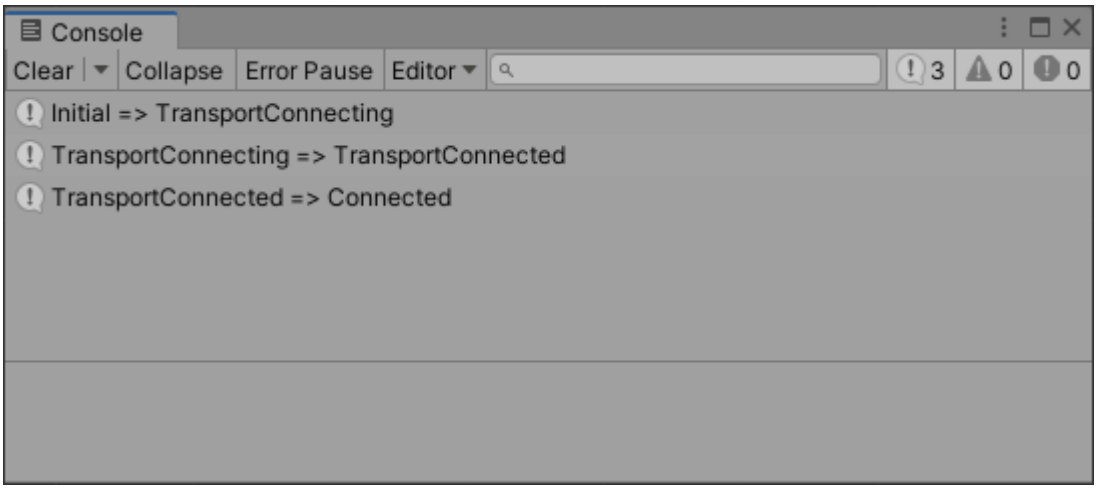
All of the event handlers have different signatures so they will be mapped to the right event.

Connecting

So far we done a basic setup of the client and added a few event handlers, but still not connected to the mqtt server. To start connecting to the server we can add a `BeginConnect` call after the client setup:

```
client.BeginConnect(ConnectPacketBuilderCallback);

private ConnectPacketBuilder ConnectPacketBuilderCallback(MQTTClient client, ConnectPacketBuilde
r builder)
{
    return builder;
}
```

`BeginConnect` expects a function that returns with a connect packet builder. The builder is used to build the MQTT connect packet **after** the transport successfully connected to the server. Through this builder we can set up basic authentication, a will, customize negotiable values like keep alive intervals and many more. For now we can leave it as is, just returning the builder received in the second parameter. Now we can test it and when run in Unity the Console should show something like this:



Note: `BeginConnect` and other functions starting with 'Begin' are non-blocking! To execute code after connected an `OnConnected` event handler must be added.

And disconnecting

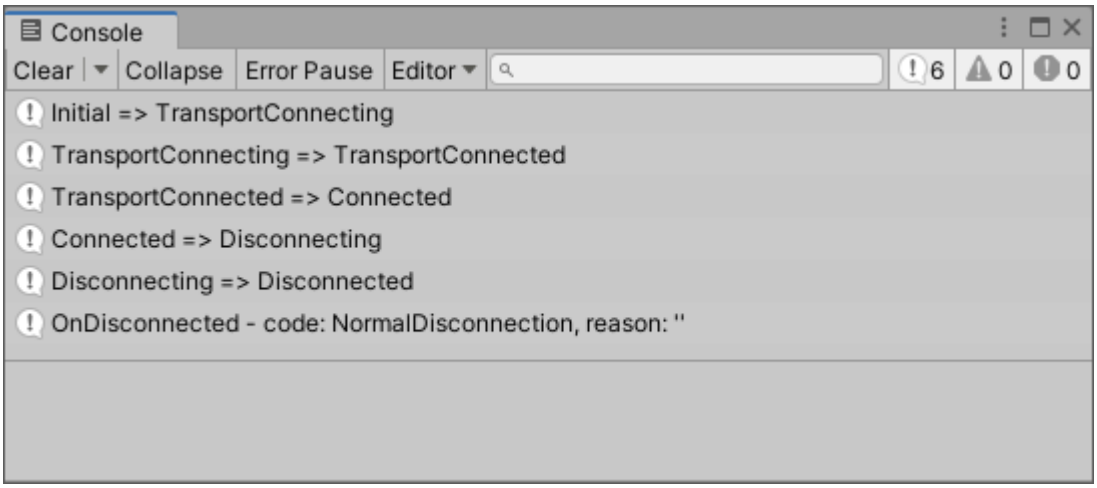
It's advised to disconnect when the MQTT client no longer needed. To disconnect, at a bare minimum we have to call the `MQTTClient`'s `CreateDisconnectPacketBuilder`, optionally call its `With*` functions then finally `BeginDisconnect` to send the disconnect packet to the server and let the plugin do its cleanup.

In this example we are going to set the reason code and send a nice message to the server too:

```
private void OnDestroy()
{
    client?.CreateDisconnectPacketBuilder()
        .WithReasonCode(DisconnectReasonCodes.NormalDisconnection)
        .WithReasonString("Bye")
        .BeginDisconnect();
}
```

The plugin heavily uses the builder pattern as there are a lot of optional fields that can be sent. This is the case with disconnection too. When `client.CreateDisconnectPacketBuilder().BeginDisconnect()` is used it's going to send a `DisconnectReasonCodes.NormalDisconnection` without any additional data.

Now entering to and exiting from play mode in Unity should generate the following output in the console:



Final code

Putting it all together, the whole file should look like something like this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using BestMQTT;
using BestMQTT.Packets.Builders;

public class MQTT : MonoBehaviour
{
    MQTTClient client;

    // Start is called before the first frame update
    void Start()
    {
        client = new MQTTClientBuilder()
                        .WithOptions(new ConnectionOptionsBuilder().WithTCP("broker.emqx.io", 18
83))
                        .WithEventHandler(OnDisconnected)
                        .WithEventHandler(OnStateChanged)
                        .WithEventHandler(OnError)
                       .CreateClient();

        client.BeginConnect(ConnectPacketBuilderCallback);
    }

    private void OnDestroy()
    {
        client?.CreateDisconnectPacketBuilder()
            .WithReasonCode(DisconnectReasonCodes.NormalDisconnection)
            .WithReasonString("Bye")
            .BeginDisconnect();
    }

    private ConnectPacketBuilder ConnectPacketBuilderCallback(MQTTClient client, ConnectPacketBu
ilder builder)
    {
        return builder;
    }

    private void OnStateChanged(MQTTClient client, ClientStates oldState, ClientStates newState)
    {
        Debug.Log($"{oldState} => {newState}");
    }

    private void OnDisconnected(MQTTClient client, DisconnectReasonCodes code, string reason)
    {
        Debug.Log($"OnDisconnected - code: {code}, reason: '{reason}'");
    }
```

```
    private void OnError(MQTTClient client, string reason)
    {
        Debug.Log($"OnError reason: '{reason}'");
    }
}
```

Publish

Summary: How to use ApplicationMessagePacketBuilder to create and send an Application Message to the MQTT server

Initial setup

Extending the code from the previous chapter, we can add new event handler `OnConnected` :

```
client = new MQTTClientBuilder()
                // ...
                .WithEventHandler(OnConnected)
                // ...
                .CreateClient();

private void OnConnected(MQTTClient client)
{
}
```

The `OnConnected` event is fired when the client successfully connected with its transport, the MQTT protocol negotiation is over and messages can be sent. We will add code to this new callback to publish a new application message.

Simple Publish

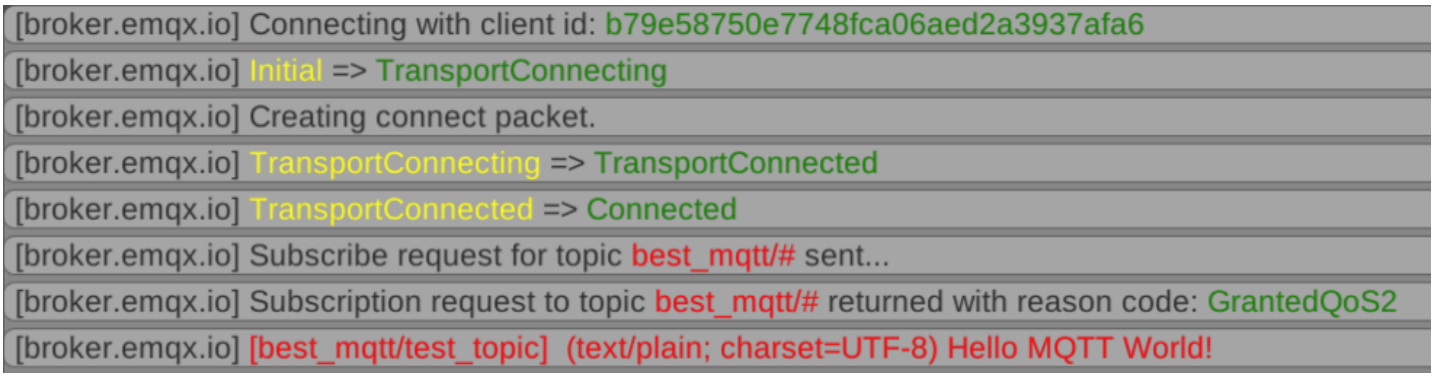
`MQTTClient` has a `CreateApplicationMessageBuilder` function to help in the creation and sending of application messages.

```
client.CreateApplicationMessageBuilder("best_mqtt/test_topic")
        .WithPayload("Hello MQTT World!")
        .BeginPublish();
```

Code	Description
`client.CreateApplicationMessageBuilder("best_mqtt/test_topic")`	Create the builder with the given topic name.
`.WithPayload("Hello MQTT World!")`	Set the payload. WithPayload has an overload to accept a byte[] too to send binary data.

Code	Description
`.BeginPublish();`	Start sending the application message.

Using another MQTT client (like the one under the WebGL demo topic (page 0)) to connect to the same broker and subscribing to the *BestMQTT/hello* topic produces the following output:



Setting QoS Levels

Best MQTT supports both sending and receiving all three QoS levels on all supported platforms, including WebGL. We can define the application message's QoS level by using the WithQoS function:

```
client.CreateApplicationMessageBuilder("best_mqtt/test_topic")
        // ...
        .WithQoS(BestMQTT.Packets.QoSLevels.ExactlyOnceDelivery) // send with QoS 2
        .BeginPublish();
```

QoS	Description
AtMostOnceDelivery (QoS 0)	Sent only once by the client, no guarantee the server receives it.
AtLeastOnceDelivery (QoS 1)	Sent at least once, until receives an acknowledgement from the server that received it. QoS 1 messages might received more than once.
ExactlyOnceDelivery (QoS 2)	The protocol makes sure that QoS 2 messages are received only once.

Note: If `.WithQoS()` call is not present the plugin sends messages with `AtMostOnceDelivery` (QoS 0).

Warning: QoS setting of an application message controls **only** the QoS level between the client and the

server. The server can deliver the message to its subscribing clients with different QoS levels!

Content-Type

Using the `WithContentType` function we can specify what type of content we are sending. It's a textual description of the content. It's usually a MIME content type, but because neither the plugin or the server processing it, its meaning and usage is defined by the sending and receiving application.

```
client.CreateApplicationMessageBuilder("best_mqtt/test_topic")
        // ...
        .WithContentType("text/plain; charset=UTF-8")
        .BeginPublish();
```

Setting Topic Alias

To send less data every time a message is published to a topic, a topic-alias can be assigned to the topic name. This way only the topic-alias number is sent instead of the full topic name possibly saving bandwith and processing time. To add a topic name mapping add the following line to the OnConnected handler before creating the application message builder:

```
client.AddTopicAlias("best_mqtt/test_topic");
```

When a topic alias added the next message published with the topic will inform the server about the mapping. Consecutive messages will use the mapping by omitting the topic name.

☑ **Tip:** Best use of a topic name mapping is when messages are sent frequently to the topic and/or the topic name is long.

Final code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using BestMQTT;
using BestMQTT.Packets.Builders;
using System;

public class MQTT : MonoBehaviour
{
    MQTTClient client;

    // Start is called before the first frame update
    void Start()
    {
        client = new MQTTClientBuilder()
                        .WithOptions(new ConnectionOptionsBuilder().WithTCP("broker.emqx.io", 18
83))
                        .WithEventHandler(OnConnected)
                        .WithEventHandler(OnDisconnected)
                        .WithEventHandler(OnStateChanged)
                        .WithEventHandler(OnError)
                      .CreateClient();

        client.BeginConnect(ConnectPacketBuilderCallback);
    }

    private void OnConnected(MQTTClient client)
    {
        client.AddTopicAlias("best_mqtt/test_topic");

        client.CreateApplicationMessageBuilder("best_mqtt/test_topic")
                .WithPayload("Hello MQTT World!")
                .WithQoS(BestMQTT.Packets.QoSLevels.ExactlyOnceDelivery)
                .WithContentType("text/plain; charset=UTF-8")
                .BeginPublish();
    }

    private void OnDestroy()
    {
        client?.CreateDisconnectPacketBuilder()
            .WithReasonCode(DisconnectReasonCodes.NormalDisconnection)
            .WithReasonString("Bye")
            .BeginDisconnect();
    }

    private ConnectPacketBuilder ConnectPacketBuilderCallback(MQTTClient client, ConnectPacketBu
ilder builder)
    {
```

```
        return builder;
    }

    private void OnStateChanged(MQTTClient client, ClientStates oldState, ClientStates newState)
    {
        Debug.Log($"{oldState} => {newState}");
    }

    private void OnDisconnected(MQTTClient client, DisconnectReasonCodes code, string reason)
    {
        Debug.Log($"OnDisconnected - code: {code}, reason: '{reason}'");
    }

    private void OnError(MQTTClient client, string reason)
    {
        Debug.Log($"OnError reason: '{reason}'");
    }
}
```

Subscribe

Learnt how to publish a message to a topic in the previous article, now it's time to learn how to subscribe to a topic and process application messages.

Subscribe

Calling `CreateSubscriptionBuilder` a topic filter (page 24) must be specified, all messages sent to a matching topic will be sent to the subscribing client. While the subscription would work without adding a message callback, usually we want to process application messages in a callback that we can add with `WithMessageCallback` :

```
client.CreateSubscriptionBuilder("best_mqtt/test_topic")
        .WithMessageCallback(OnMessage)
        .BeginSubscribe();

private void OnMessage(MQTTClient client, SubscriptionTopic topic, string topicName, Application
Message message)
{
    // Convert the raw payload to a string
    var payload = Encoding.UTF8.GetString(message.Payload.Data, message.Payload.Offset, messag
e.Payload.Count);

    Debug.Log($"Content-Type: '{message.ContentType}' Payload: '{payload}'");
}
```

In this example we subscribe to a concrete topic 'best_mqtt/test_topic' without using a wildcard. Any message sent to this topic the server will forward to the subscribing client.

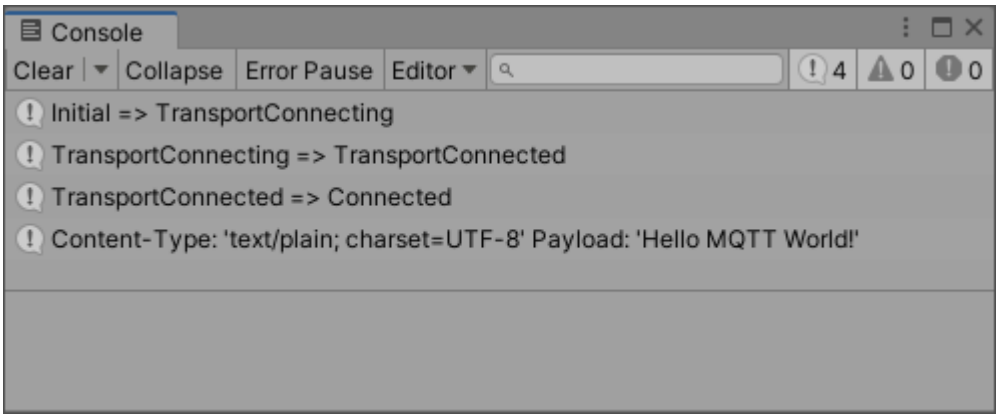
The `OnMessage` callback has the following parameters:

Type	Name	Description
`MQTTClient`	`client`	The `MQTTClient` instance that we created the subscription with.
`SubscriptionTopic`	`topic`	A `SubscriptionTopic` instance that contains the original topic filter that the subscription subscribed to.
`string`	`topicName`	The topic name that matched with the topic filter. Because the topic filter can have wildecard `topicName` can be different from the topic filter the subscription created with.
`string`	`message`	The application message sent by the server. Among other fields, it has the `Payload` and `ContentType` fields we used in the previous article.

> **Warning:** Do not keep a reference to the message's payload, it will be recycled after the event handler!

Because MQTT packets (subscription, application messages, etc.) are sent and processed in order, the subscription request is received by the server first and right after the application message too. The application message's topic matches with the subscription's topic filter so the server will send back the application message to our client.

```
private void OnConnected(MQTTClient client)
{
    client.CreateSubscriptionBuilder("best_mqtt/test_topic")
            .WithMessageCallback(OnMessage)
            .BeginSubscribe();

    client.CreateApplicationMessageBuilder("best_mqtt/test_topic")
            .WithPayload("Hello MQTT World!")
            .WithQoS(BestMQTT.Packets.QoSLevels.ExactlyOnceDelivery)
            .WithContentType("text/plain; charset=UTF-8")
            .BeginPublish();
}
```



Define Maximum QoS

Each subscription can define its supported QoS level (page 26) that the server can deliver application messages with. This is the maximum level the client wants to support for that subscription, but the server can choose a lower maximum too.

```
client.CreateSubscriptionBuilder("best_mqtt/test_topic")
        // ...
        .WithMaximumQoS(QoSLevels.ExactlyOnceDelivery)
        .BeginSubscribe();
```

Note: If there's no `.WithMaximumQoS` call, the plugin uses the server's maximum supported QoS level.

Acknowledgement callback

In some cases we might want to know when and how successfully the subscribe operation gone. For example whether the subscription succeeded or not, or what QoS level is granted by the server. For these cases we can add an acknowledgement callback:

```
client.CreateSubscriptionBuilder("best_mqtt/test_topic")
                // ...
        .WithAcknowledgementCallback(OnSubscriptionAcknowledged)
        .BeginSubscribe();

private void OnSubscriptionAcknowledged(MQTTClient client, SubscriptionTopic topic, SubscribeAck
ReasonCodes reasonCode)
{
    if (reasonCode <= SubscribeAckReasonCodes.GrantedQoS2)
        Debug.Log($"Successfully subscribed with topic filter '{topic.Filter.OriginalFilter}'. Q
oS granted by the server: {reasonCode}");
    else
        Debug.Log($"Could not subscribe with topic filter '{topic.Filter.OriginalFilter}'! Erro
r code: {reasonCode}");
}
```

`reasonCode` is a success/error code. If it's less than or equal to `SubscribeAckReasonCodes.GrantedQoS2` the client successfully subscribed with the given topic filter. Otherwise `reasonCode` is an error code describing why the subscription attempt failed.

Bulk subscribe

It's possible to subscribe to multiple topics in one go. Using `MQTTClient`'s `CreateBulkSubscriptionBuilder`, `WithTopic` can be used multiple times and the client will send the subscribe request in one packet. `SubscribeTopicBuilder` has the same options to set as the builder returned with `CreateSubscriptionBuilder`.

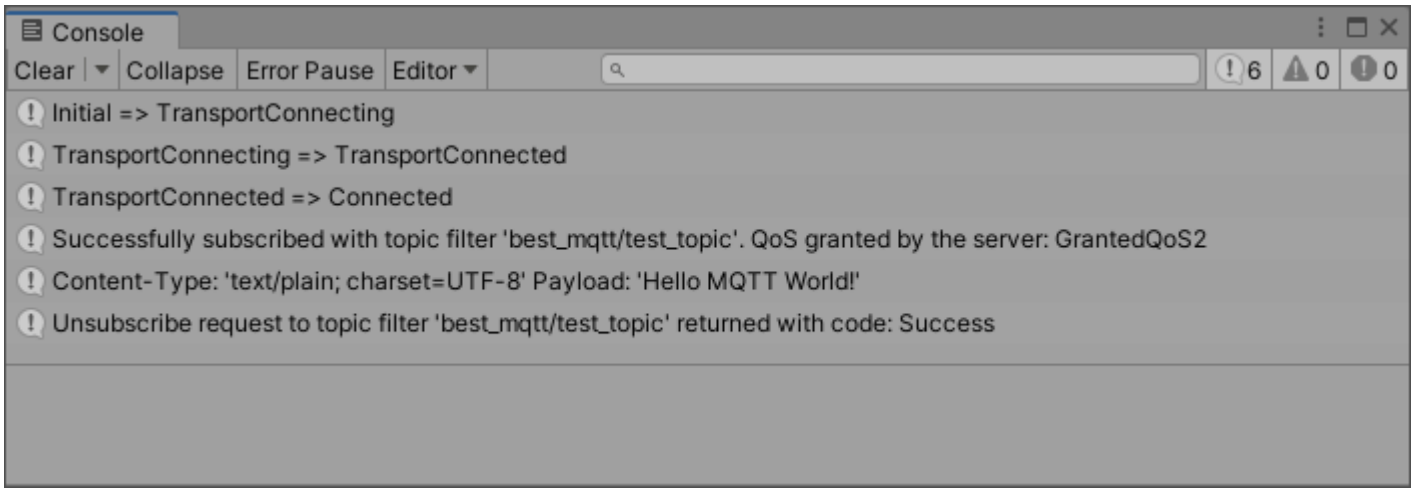
```
client.CreateBulkSubscriptionBuilder()
    .WithTopic(new SubscribeTopicBuilder("best_mqtt/topic_1")
                        .WithMessageCallback(OnTopic1Message))
    .WithTopic(new SubscribeTopicBuilder("best_mqtt/topic_2")
                        .WithMessageCallback(OnTopic2Message))
    .BeginSubscribe();
```

Unsubscribe

To unsubscribe from a topic filter the `CreateUnsubscribePacketBuilder` / `CreateBulkUnsubscribePacketBuilder` functions can be used, similarly as theirs subscription counterparts:

```
client.CreateUnsubscribePacketBuilder("best_mqtt/test_topic")
    .WithAcknowledgementCallback((client, topicFilter, reasonCode) => Debug.Log($"Unsubscribe re
quest to topic filter '{topicFilter}' returned with code: {reasonCode}"))
    .BeginUnsubscribe();
```

Adding the above code to the `OnMessage` callback to unsubscribe from the topic after a message is received produces the following output:



Final code

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using BestMQTT;
using BestMQTT.Packets.Builders;
using BestMQTT.Packets;
using System.Text;

public class MQTT : MonoBehaviour
{
    MQTTClient client;

    // Start is called before the first frame update
    void Start()
    {
        client = new MQTTClientBuilder()
#if !UNITY_WEBGL || UNITY_EDITOR
                        .WithOptions(new ConnectionOptionsBuilder().WithTCP("broker.emqx.io", 18
83))
#else
                        .WithOptions(new ConnectionOptionsBuilder().WithWebSocket("broker.emqx.i
o", 8084).WithTLS())
#endif
                        .WithEventHandler(OnConnected)
                        .WithEventHandler(OnDisconnected)
                        .WithEventHandler(OnStateChanged)
                        .WithEventHandler(OnError)
                      .CreateClient();

        client.BeginConnect(ConnectPacketBuilderCallback);
    }

    private void OnConnected(MQTTClient client)
    {
        client.AddTopicAlias("best_mqtt/test_topic");

        client.CreateSubscriptionBuilder("best_mqtt/test_topic")
                .WithMessageCallback(OnMessage)
                .WithAcknowledgementCallback(OnSubscriptionAcknowledged)
                .WithMaximumQoS(QoSLevels.ExactlyOnceDelivery)
                .BeginSubscribe();

        client.CreateApplicationMessageBuilder("best_mqtt/test_topic")
                .WithPayload("Hello MQTT World!")
                .WithQoS(QoSLevels.ExactlyOnceDelivery)
                .WithContentType("text/plain; charset=UTF-8")
                .BeginPublish();
```

```
    }

    private void OnMessage(MQTTClient client, SubscriptionTopic topic, string topicName, Applica
tionMessage message)
    {
        // Convert the raw payload to a string
        var payload = Encoding.UTF8.GetString(message.Payload.Data, message.Payload.Offset, mess
age.Payload.Count);

        Debug.Log($"Content-Type: '{message.ContentType}' Payload: '{payload}'");

        client.CreateUnsubscribePacketBuilder("best_mqtt/test_topic")
            .WithAcknowledgementCallback((client, topicFilter, reasonCode) => Debug.Log($"Unsubs
cribe request to topic filter '{topicFilter}' returned with code: {reasonCode}"))
            .BeginUnsubscribe();
    }

    private void OnSubscriptionAcknowledged(MQTTClient client, SubscriptionTopic topic, Subscrib
eAckReasonCodes reasonCode)
    {
        if (reasonCode <= SubscribeAckReasonCodes.GrantedQoS2)
            Debug.Log($"Successfully subscribed with topic filter '{topic.Filter.OriginalFilte
r}'. QoS granted by the server: {reasonCode}");
        else
            Debug.Log($"Could not subscribe with topic filter '{topic.Filter.OriginalFilter}'! E
rror code: {reasonCode}");
    }

    private void OnDestroy()
    {
        client?.CreateDisconnectPacketBuilder()
            .WithReasonCode(DisconnectReasonCodes.NormalDisconnection)
            .WithReasonString("Bye")
            .BeginDisconnect();
    }

    private ConnectPacketBuilder ConnectPacketBuilderCallback(MQTTClient client, ConnectPacketBu
ilder builder)
    {
        return builder;
    }

    private void OnStateChanged(MQTTClient client, ClientStates oldState, ClientStates newState)
    {
        Debug.Log($"{oldState} => {newState}");
    }

    private void OnDisconnected(MQTTClient client, DisconnectReasonCodes code, string reason)
    {
```

```
        Debug.Log($"OnDisconnected - code: {code}, reason: '{reason}'");
    }

    private void OnError(MQTTClient client, string reason)
    {
        Debug.Log($"OnError reason: '{reason}'");
    }
}
```

Topic Names & Filters

Topic Names

A topic name is an UTF-8 string containing one or more levels separated by a forward slash character (`/`). A topic name must not contain any wildcard characters (`+` and `#`)! Topic names are usabe both in publish and subscribe operations.

Example topic names:

Topic Name
`topic`
`topic name`
`multi/level/topic`
`multi/level/topic/`
`/multi/level/topic/`
`/`

Note: Topic matching is case sensitive, **TOPIC** and **topic** are two different topic names!

Topic Filters

Wildcard characters can be used in topic filters to match one or more levels within a topic. MQTT has two types of wildcard characters a Single-level wildcard (`+`) and a Multi-level wildcard (`#`).

Single-level wildcard

A single-level wildcard can be used to match one level and multiple single-level wildcards can be used in one topic filter.

Topic Filter	Matching topics	Non-matching topics
`+`	`finance`	
`/+`	`/finance`	
`+/+`	`/finance`	

Topic Filter	Matching topics	Non-matching topics
`sport/+`	`sport/`	`sport`
`sport/tennis/+`	`sport/tennis/player1`	`sport/tennis/player1/ranking`
	`sport/tennis/player2`	

❗ **Warning: sport+** is not a valid topic filter because it must be used to match a whole level!

Multi-level wildcard

A multi-level wildcard (`#`) can be used to match any level in a topic name. `#` matches its parent level and any number of child levels. When `#` is used it must be the last character of the topic filter.

Topic Filter	Matching topics
`sport/#`	`sport`
`sport/tennis/player1/#`	`sport/tennis/player1`
	`sport/tennis/player1/`
	`sport/tennis/player1/ranking`
	`sport/tennis/player1/score/wimbledon`

Mixed topic filters

Single and multi level wildcards can be used in one topic filter:

Topic Filter	Matching topics
`+/tennis/#`	`sport/tennis/player1/ranking`

Quality of Service Levels

A QoS level is an agreement between a sender and a receiver about the delivery guarantees of application messages.

AtMostOnceDelivery (QoS 0)

With QoS 0 delivery of application messages are not guaranteed. The sender sends the application message, but no acknowledgement message is sent by the receiver whether it received it or not, hence no retry mechanism either. This is the fastest but most unreliable QoS level.

AtLeastOnceDelivery (QoS 1)

With QoS 1 the protocol guarantees that the receiver receives application messages at least once. This guarantee is achieved by listening for an acknowledgement message and if not received while the client is online, the sender resends the application message with the Dupplicate flag set to true when client next time goes online.

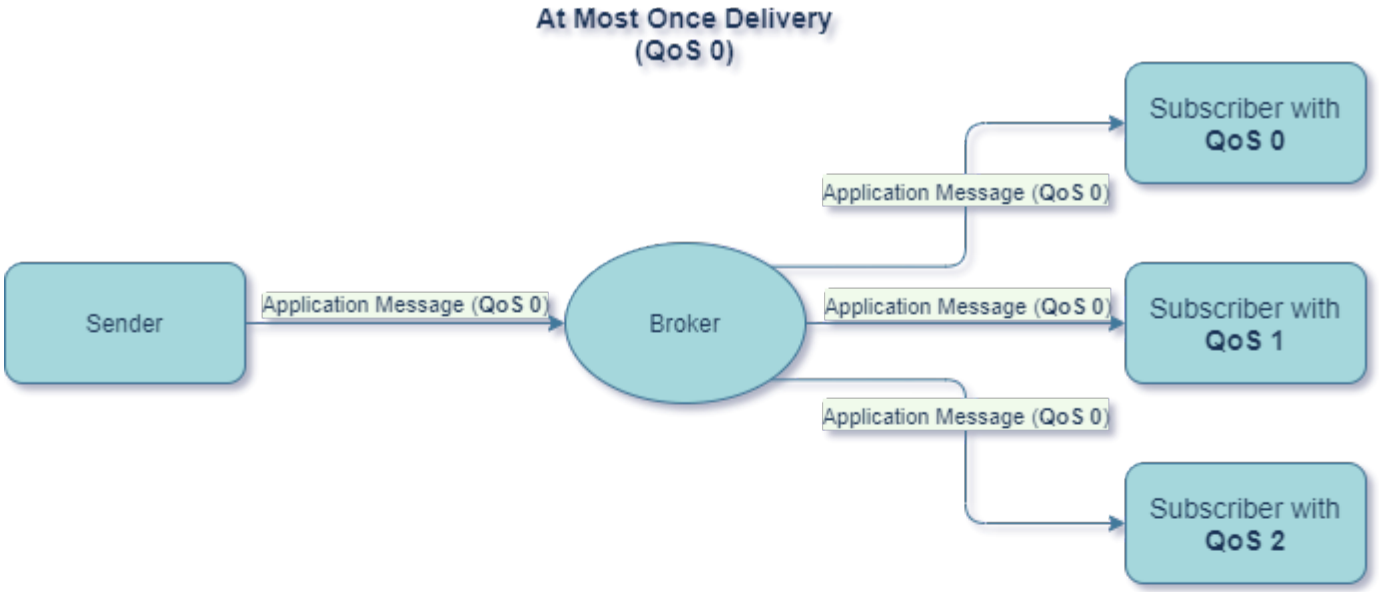
> ❗ **Warning:** Using QoS 1 the subscriber must be prepared that messages might received more than once!

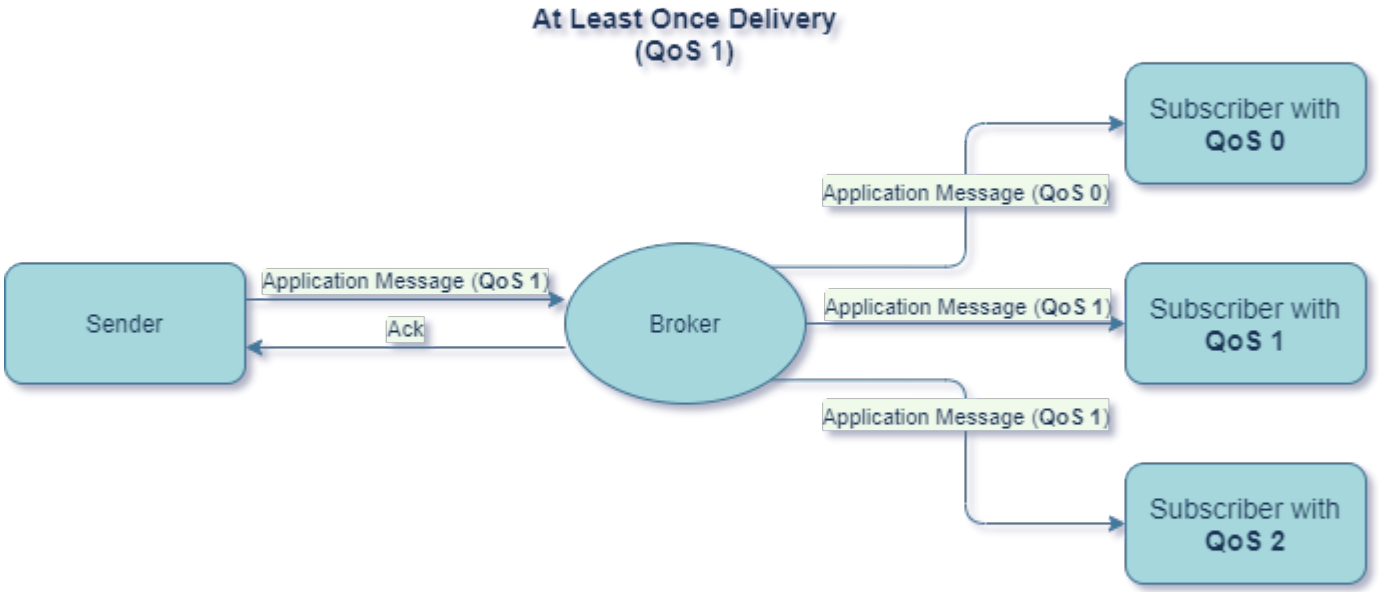
ExactlyOnceDelivery (QoS 2)

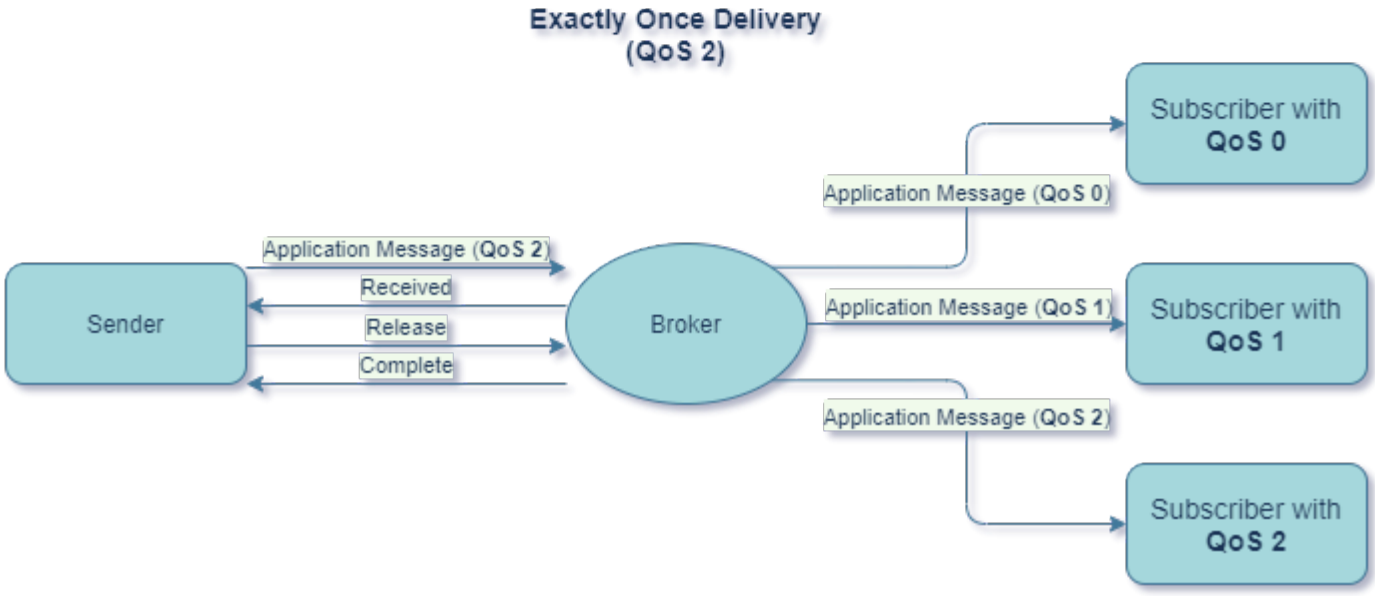
With QoS 2 the protocol guarantees that the receiver receives application messages exactly once. To achieve this reliability, MQTT v5 uses a four message handshake process that starts with the publishing of the application message. This is the most reliable but slowest QoS level.

QoS Downgrade

Clients can subscribe to a topic with their own QoS level preference and it can be lower than clients sending application messages with.


At Most Once Delivery
(QoS 0)
Sender
Application Message (QoS 0)
Broker
Application Message (QoS 0)
Application Message (QoS 0)
Application Message (QoS 0)
Subscriber with QoS 0
Subscriber with QoS 1
Subscriber with QoS 2


At Least Once Delivery
(QoS 1)
Sender
Application Message (QoS 1)
Ack
Broker
Application Message (QoS 0)
Application Message (QoS 1)
Application Message (QoS 1)
Subscriber with QoS 0
Subscriber with QoS 1
Subscriber with QoS 2


Exactly Once Delivery
(QoS 2)
Sender
Application Message (QoS 2)
Received
Release
Complete
Broker
Application Message (QoS 0)
Application Message (QoS 1)
Application Message (QoS 2)
Subscriber with QoS 0
Subscriber with QoS 1
Subscriber with QoS 2

Further reading

A very good article about QoS Levels (and many more) can be found on HiveMQ's blog .

Last-Will

Summary: A last will is a regular application message, but the client can set it up and send while connecting to the broker. The broker will distribute this last-will only when the client goes offline and the last-will's Delay Interval expires.

Creating a Last-Will

A last-will can be created in the connect packget builder callback (page 5). The following code creates a last-will that will be published to the “client/last-will” topic 60 seconds after the client gone offline.

```
private ConnectPacketBuilder ConnectPacketBuilderCallback(MQTTClient client, ConnectPacketBuilde
r builder)
{
    return builder.WithLastWill(new LastWillBuilder()
                                .WithDelayInterval(60)
                                .WithTopic("client/last-will")
                                .WithQoS(QoSLevels.ExactlyOnceDelivery)
                                .WithPayload("This is my last will!"));
}
```

Sessions

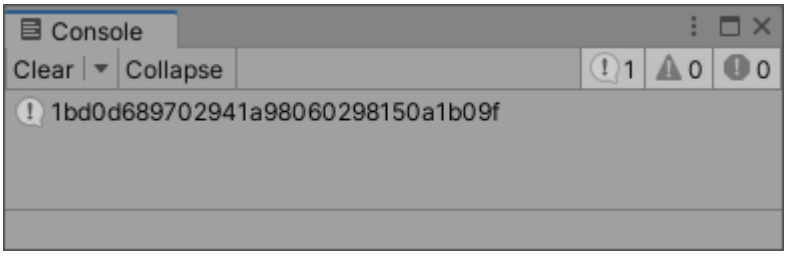
To be able to support QoS 1 and 2 both the client and the broker must have a way to store state information. Best MQTT handles session creation and resuming to the last used one automatically. With the help of the `SessionHelper` class it's possible to manage and set sessions manually.

> ❗ **Warning:** While the server stores the clients' subscriptions, Best MQTT can't recreate the binding between a topic filter and its callback. It's highly advised to always subscribe to topic filters when connected!

Get a session

`SessionHelper.Get` returns with the last used session for the given host.

```
var session = SessionHelper.Get("broker.emqx.io");
Debug.Log(session.ClientId);
```

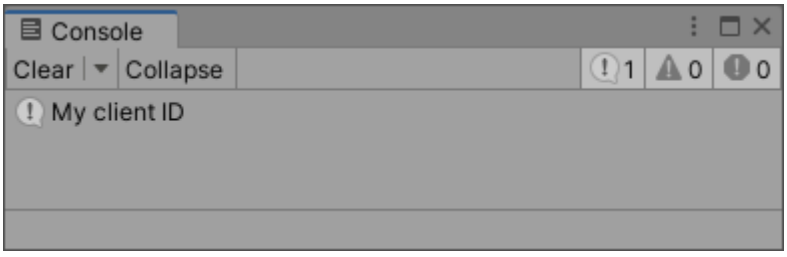


It can be used any time. If no session created yet for the host, it creates one.

Create a session with a concrete Client ID

`SessionHelper.Get` has a second, optional `clientId` parameter. If omitted returns with the last used session. If present tries to load session with that ID, and if not found creates and returns with a new one.

```
var session = SessionHelper.Get("broker.emqx.io", "My client ID");
Debug.Log(session.ClientId);
```



❗ **Warning:** A client ID must be unique across the connecting clients and the same client should use the same ID for consecutive connections!

How to use a session

```
private ConnectPacketBuilder ConnectPacketBuilderCallback(MQTTClient client, ConnectPacketBuilde
r builder)
{
    var session = SessionHelper.Get(client.Options.Host);
    return builder.WithSession(session);
}
```

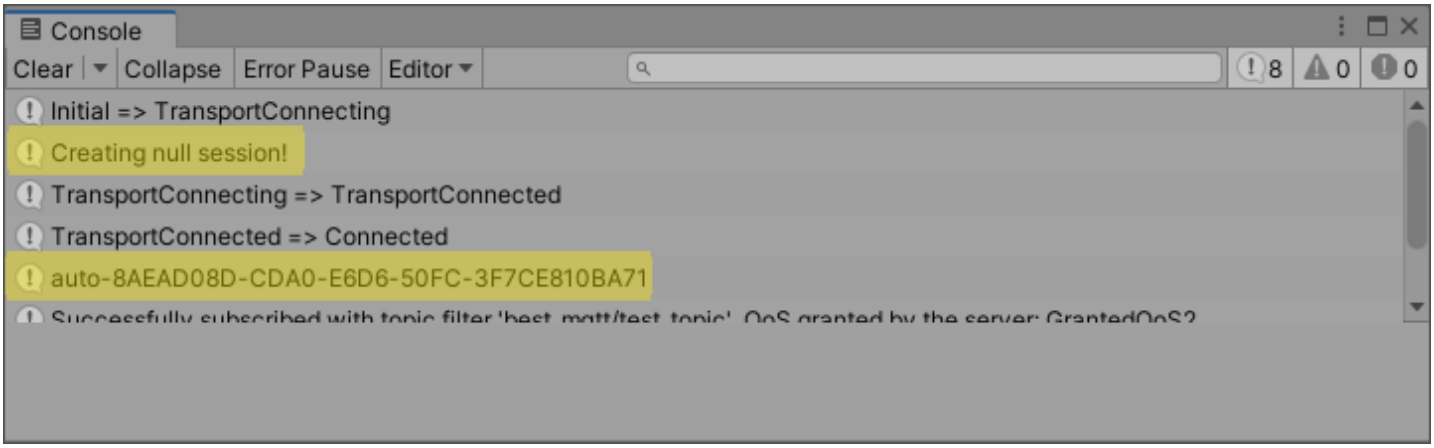
Null Sessions

To let the server assign an ID to the client, we can create and connect with a null session. In case of a null session the client will not generate and send an ID, instead expects one from the server. When the client ID is received from the server, the plugin will create a real session.

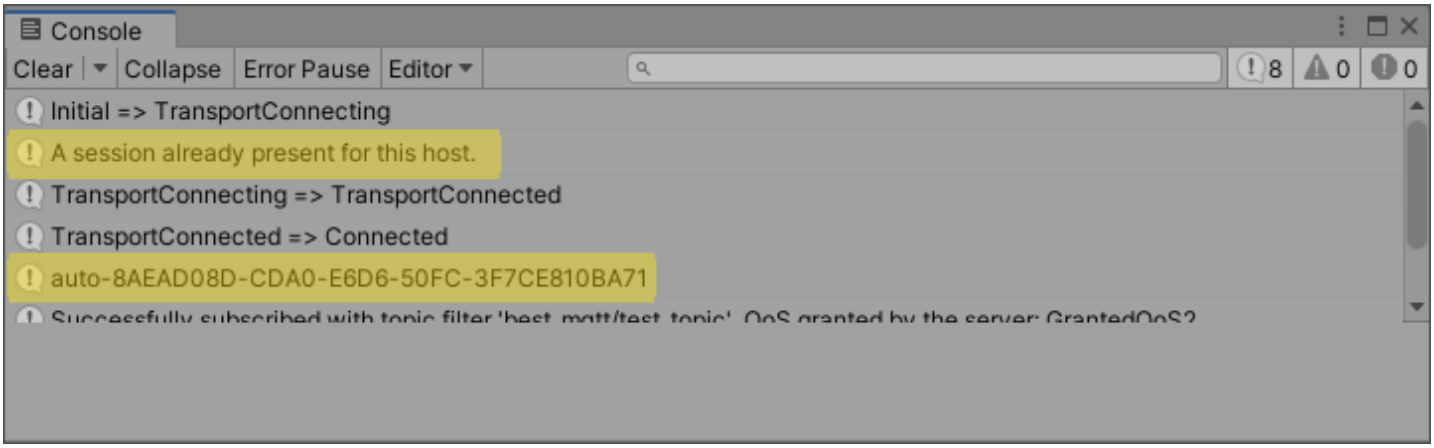
```
private ConnectPacketBuilder ConnectPacketBuilderCallback(MQTTClient client, ConnectPacketBuilde
r builder)
{
    var host = client.Options.Host;

    if (!SessionHelper.HasAny(host))
    {
        Debug.Log("Creating null session!");
        builder = builder.WithSession(SessionHelper.CreateNullSession(host));
    }
    else
        Debug.Log("A session already present for this host.");

    return builder;
}

private void OnConnected(MQTTClient client)
{
    Debug.Log(SessionHelper.Get(client.Options.Host).ClientId);

    // ...
}
```

`SessionHelper.HasAny` returns `true` if there's any session for the given host.

If this is the first time the client connects to the host, it produces an output like this:



Running the script again will skip creating a new null session and will use the one created in the previous run.



The client used the same client ID (`auto-8AEAD08D-CDA0-E6D6-50FC-3F7CE810BA71`) that received from the server previously.

Optimization Tips & Tricks

Summary: Collection of tips and tricks to optmize various aspects of the plugin.

Wrap multiple calls with `PacketBufferHelper`

Using `PacketBufferHelper` we can buffer up MQTT packets and send them in fewer network packets:

```
private void OnConnected(MQTTClient client)
{
    using (new PacketBufferHelper(client))
    {
        client.AddTopicAlias("best_mqtt/test_topic");

        client.CreateSubscriptionBuilder("best_mqtt/test_topic")
                .WithMessageCallback(OnMessage)
                .BeginSubscribe();

        client.CreateApplicationMessageBuilder("best_mqtt/test_topic")
                .WithPayload("Hello MQTT World!")
                .WithQoS(BestMQTT.Packets.QoSLevels.ExactlyOnceDelivery)
                .WithContentType("text/plain; charset=UTF-8")
                .BeginPublish();
    }
}
```

Use Topic Aliases

To spare sending the topic name every time with an application message a topic alias can be added. It's recommended to use it for long or frequantly used topic names.

```
private void OnConnected(MQTTClient client)
{
    client.AddTopicAlias("best_mqtt/test_topic");
}
```

`AddTopicAlias` doesn't generate a packet, the mapping is sent with the next application message that going to be sent with that topic name.

Compress text payloads

Large text payloads should be sent compressed when possible. Using the `WithContentType` the publisher can indicate that the payload is compressed and subscribers can act accordingly.

MQTTClient