Swift App 進階教師專班

AI 機器學習與設備(BLE & IOT)通訊

Day 4 – Bluetooth 通訊 Central App (Part 2)

IOT 通訊 MQTT Client App

主辦：新北市政府教育局
日期：2023.07.27 (四)
講師：友教有限公司 Michael

Bluetooth Central App – part 2 1

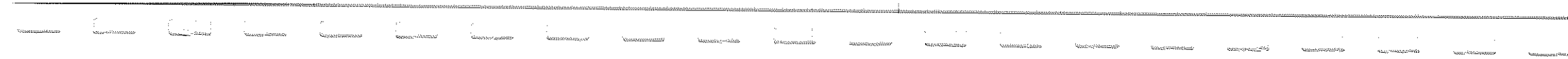
CBPeripheralDelegate 協議 3
處理並儲存藍牙資料 10
畫面顯示藍牙接收資料 11

MQTT 基本觀念 13

Broker (MQTT Server) and Topic 13
Subscriber and Publisher (MQTT Client) 14
MQTT Quality of Service (QoS) 15
EasyMQTT (App) 15

MQTT Client App 19

Closure 語法處理 MQTT 訊息 – LightMQTT 19
Delegate 模式處理 MQTT 訊息 – SwiftMQTT 27



Bluetooth Central App – part 2

先回顧一下前一堂課的內容：

在 Bluetooth Central App 專案中，我們會定義一個 class BLEManager 來負責處理所有藍牙相關的操作，其中這個 BLEManager 需要遵循 CBCentralManagerDelegate 和 CBPeripheralDelegate 兩個協議，並在 ContentView 畫面呈現藍芽接收到的資料。

前一堂課已經完成 Bluetooth Central App 這個專案的前置作業與 A ~ B 步驟了，讓藍牙成功連線與斷開連線，接下來我們要處理藍牙 peripheral 的資料。

下方是程式區塊對應圖：

```
4   class BLEManager: NSObject, ObservableObject, CBCentralManagerDelegate, CBPeripheralDelegate {
    Type 'BLEManager' does not conform to protocol 'CBCentralManagerDelegate'
5       let peripheralName = "ZenNano_101" //記得要修改為正確的名稱
6       let serviceID = "19B10010-E8F2-537E-4F6C-D104768A1214"
7       let stepCharID = "19B10011-E8F2-537E-4F6C-D104768A1214"    //step
8       let tempCharID = "19B10012-E8F2-537E-4F6C-D104768A1214"     //temperature
9       // @Published 變數
10      @Published var isSwitchedOn = false
11
12      //宣告 centralManager 與 peripheral
13      var centralManager: CBCentralManager!
14      var peripheral: CBPeripheral!
15
16      //初始化 centralManager
17
18      //CBCentralManagerDelegate 協議：設定6個函數
19
20      //CBPeripheralDelegate 協議：設定3個函數
21
22  }
23
24  struct ContentView: View {
25      @ObservedObject var bleManager = BLEManager()
26      var body: some View {
27          VStack(alignment: .leading, spacing: 20) {
28              Text("Bluetooth : " + (bleManager.isSwitchedOn ? "ON" : "OFF"))
29          }
30      }
31  }
```

@Published 變數宣告區

步驟 B ~E

步驟 E

下方將 B ~ E的步驟列出，雖然B的步驟都已經完成，但是在接下來的內容，我們會需修改B裡面的函數：

步驟B. 為了符合 CBCentralManagerDelegate 協議，與 central 設備需要執行的內容，需要設定 6 個函數，分別為：

執行順序	呼叫	函數名	功能說明
B1	被動	centralManagerDidUpdateState(_:)	確認 central 藍牙已經開啟。
B2	主動	startScanning()	設定 central 設備的掃描函數。
B3	被動	centralManager(_:didDiscover:advertisementData:rssi:)	Central 掃描 peripheral，判斷 peripheral.name 為要連線的裝置名稱後建立連線。
B4	被動	centralManager(_:didConnect:)	成功連線後，發現 peripheral 的 Services。
B5	主動	disconnect()	設定與 peripheral 斷開連線。
B6	被動	centralManager(_:didDisconnectPeripheral:error:)	與 peripheral 斷開連線後要執行的內容。

步驟C. 符合 CBPeripheralDelegate 協議，設定 3 個函數，分別為：

執行順序	呼叫	函數名	功能說明
C1	被動	peripheral(_:didDiscoverServices:)	發現並確認 peripheral 的 Services ID 相符後，發現 peripheral 的 Characteristics。
C2	被動	peripheral(_:didDiscoverCharacteristicsFor:error:)	發現 peripheral 的 Characteristics ID 相符後，設定 Notify Characteristics 的值。
C3	被動	peripheral(_:didUpdateValueFor:error:)	若Characteristics 的值更新後，要做相應的處理。

步驟D. 修改 C3 函數，將收到的 peripheral 資料處理後儲存，修改 B6 函數。

步驟E. 將收到的資料呈現在 ContentView 畫面內 ，修改 B6 函數使斷開連結後，畫面的步數與溫度資料清空。

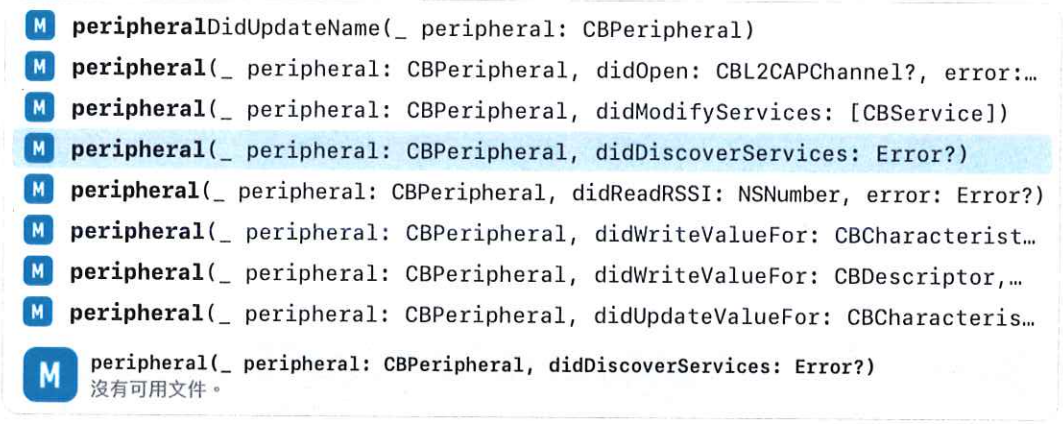
CBPeripheralDelegate 協議

C1 程式碼：peripheral(_:didDiscoverServices:) 函數

在 class BLEManager 內，設定 peripheral(_:didDiscoverServices:) 這一個函數，當 central 與 peripheral 連線之後，發現 peripheral 的服務 services 之後要執行的內容。

使用選單加入 peripheral(_:didDiscoverServices:) ：

```
func peripheral
```



加入的函數如下：

```
func peripheral(_ peripheral: CBPeripheral, didDiscoverServices error:
 Error?) {

}
```

在 B4 程式碼 centralManager-didConnect 的函數內，執行發現 peripheral 的 services 之後，peripheral 的 services 是一個 array 。

```
func centralManager(_ central: CBCentralManager, didConnect peripheral:
 CBPeripheral) {
    isConnected = true
    self.peripheral.discoverServices([CBUUID(string: serviceID)])
}
```

(已定義函數)

在目前這個函數內，我們需要找到 peripheral.services 這一個 Array 中，符合我們要的 serviceID ，之後再進行發現這個 serviceID 底下的 characteristics 。

使用 for 迴圈執行 peripheral.services ，讀取其 Array 中每一個 peripheral.services 的內容，如下：

```
63      func peripheral(_ peripheral: CBPeripheral, didDiscoverServices error:
         Error?) {
64          for service in peripheral.services {
```

For-in loop requires '[CBService]?' to conform to 'Sequence'; did you mean to unwrap optional?

```
65          }
66      }
```

此時會出現一個錯誤，需要在 peripheral.services 後面加一個！，如下圖。

因為它的型別為 [CBServices]? 是一個 optional 型別，在還沒發現 peripheral.services 時，[CBServices] 不存在，此時值為 nil ，所以必須為一個 optional 。

```
func peripheral(_ peripheral: CBPeripheral, didDiscoverServices error:
 Error?) {
    for service in peripheral.services! {
        if service.uuid == CBUUID(string: serviceID) {

        }
    }
}
```

peripheral.service 一定會是有內容時，此函數才會被呼叫，所以不用擔心使用！之後會遇到 App 當機的問題。

接下來紅色方框內，讀取每一個 peripheral.services 時，都會使用 if service.uuid == CBUUID(string: serviceID) { } 來判斷這一個 services 的 uuid 是否為 serviceID 。

如果是才進行發現服務的特徵，使用 self.peripheral 的方法 .discoverCharacteristics()：

```
self.peripheral.discoverCharacteristics(characteristicUUIDs: [CBUUID]?, for: CBService)
```

.discoverCharacteristics() 內，第一個參數要放一個 Array 格式為 CBUUID ，內容為需要讀取的 characteristics ID ，如下：

```
[CBUUID(string: stepCharID), CBUUID(string: tempCharID)]
```

第二個參數為目前這一個 services 。

完成的 function 如下：

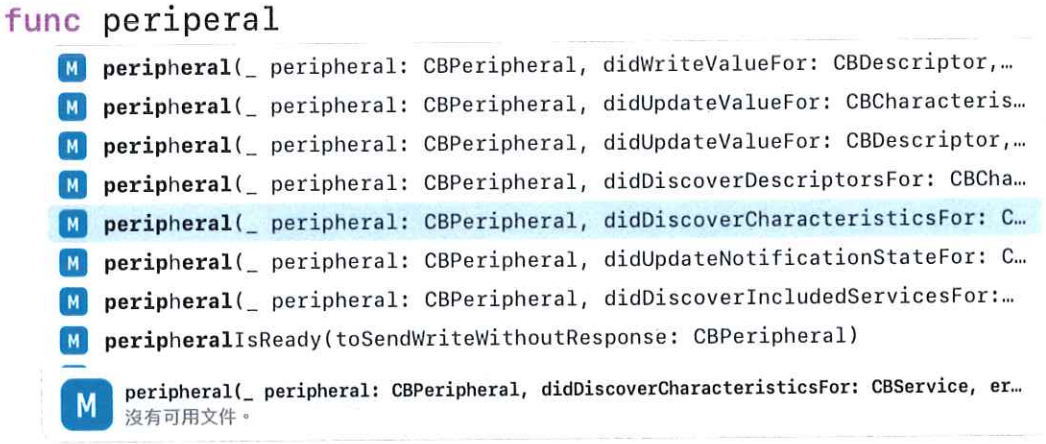
```
func peripheral(_ peripheral: CBPeripheral, didDiscoverServices error:
 Error?) {
    for service in peripheral.services! {
        if service.uuid == CBUUID(string: serviceID) {
            self.peripheral.discoverCharacteristics([CBUUID(string:
             stepCharID), CBUUID(string: tempCharID)], for: service)
        }
    }
}
```

語法筆記：

C2 程式碼：peripheral(_:didDiscoverCharacteristicsFor:error:) 函數

在 class BLEManager 內，設定 peripheral(_:didDiscoverCharacteristicsFor:error:) 這一個函數，當發現 peripheral 服務 services 的 characteristics 之後要執行的內容。

使用選單加入 peripheral(_:didDiscoverCharacteristicsFor:error:)：



加入的函數如下：

```
func peripheral(_ peripheral: CBPeripheral, didDiscoverCharacteristicsFor
 service: CBService, error: Error?) {

}
```

這個函數被自動呼叫時，我們已經發現了 peripheral 的 service 底下的 characteristics ，這個 service.characteristics 是一個 Array 。

我們定義一個 targetIDs 陣列 ，內容放我們感興趣的 characteristics ID：

```
let targetIDs = [CBUUID(string: stepCharID), CBUUID(string:
tempCharID)]
```

使用 for 迴圈執行 service.characteristics，讀取其 Array 中每一個 characteristic 的內容，記得 service.characteristics 後方需要加！，原因跟前一個函數一樣。如下：

```
for characteristic in service.characteristics! {

}
```

這個函數只有在 service.characteristics 有內容的時候才會被呼叫，所以不用擔心 service.characteristics 遇到空值 nil 的狀況。

接下來要檢查 service.characteristics 與 targetIDs 相同時，對該特徵設定屬性為監聽 .setNotifyValue(_:for:)，若值發生變化，會主動通知。

完整的函數如下：

```
func peripheral(_ peripheral: CBPeripheral, didDiscoverCharacteristicsFor service:
 CBService, error: Error?) {
    let targetIDs = [CBUUID(string: stepCharID), CBUUID(string: tempCharID)]
    for characteristic in service.characteristics! {
        if targetIDs.contains(characteristic.uuid) {
            self.peripheral.setNotifyValue(true, for: characteristic)
        }
    }
}
```

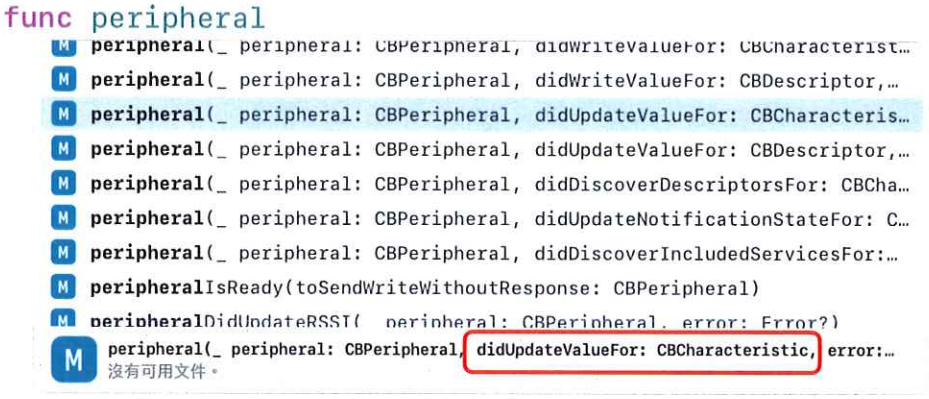
比較的方法使用 targetIDs.contains(characteristic.uuid) ，這個意思是 targetIDs 若包含 characteristic.uuid 則回傳 true，並執行 self.peripheral.setNotifyValue(true, for: characteristic) ，將此 characteristic 設為監聽屬性。

語法筆記：

C3 程式碼：peripheral(_:didUpdateValueFor:error:) 函數

在 class BLEManager 內，設定 peripheral(_:didUpdateValueFor:error:) 這一個函數，當我們關注的 characteristics 屬性已設定為監聽後，接下來要處理收到的內容，再將其呈現在畫面上。

使用選單加入 peripheral(_:didUpdateValueFor:error:) ：



注意：選單裡有兩個非常相似的函數，記得要選 didUpdateValueFor 型別為 CBCharacteristic的這個函數。

加入的函數如下：

監聽 characteristics 的數值 (value) 變化，如果收到數值有變化就會自動呼叫這個函數來

```
func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor characteristic:
 CBCharacteristic, error: Error?) {

}
```

處理資料。

首先，使用 if let 來確保 characteristics.value 不為空值時，才進行處理資料，並將它存為 data 這個變數，在後面處理資料時使用：

```
func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor characteristic:
 CBCharacteristic, error: Error?) {
    if let data = characteristic.value {
        let dataString = data.map{ String(format: "%d", $0) }.joined()

    }
}
```

Initial

接著將 data 轉檔為十進位的字串資料 dataString。
我們要監聽的這兩個 characteristics.value 內容都為數字，所以可以直接轉檔為數字字串。
data.map { String(format: "%d", $0) } 將 data 使用 map 函數，將 data 內的每一個元素轉檔，轉檔格式使用 String(format: "%d", $0) 。

.joined() 這個函數是將 map 產生的字串陣列串接起來，形成一個單一的字串。

我們監聽了兩個 characteristics.value，所以 dataString 收到的內容有可能為步數資料或是溫度資料。

先使用不同的 characteristics.uuid 來將資料分開，並分別在主控台使用 print 印出來 dataString 的內容：

```
func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor characteristic:
 CBCharacteristic, error: Error?) {
    if let data = characteristic.value {
        let dataString = data.map{ String(format: "%d", $0) }.joined()
        if characteristic.uuid == CBUUID(string: stepCharID) {
            print("Step data: " + dataString)
        } else if characteristic.uuid == CBUUID(string: tempCharID) {
            print("Temperature data: " + dataString)
        }
    }
}
```

如果 characteristics.uuid 與 stepCharID 相同，則收到的 dataSting 為步數資料，若 characteristics.uuid 與 tempCharID 相同，則收到的 dataSting 為溫度資料。

測試藍牙設備連線，看看在主控台是否能印出下面的資料：

```
Temperature data: 31
Step data: 1
Temperature data: 31
Step data: 1
Temperature data: 31
Step data: 0
```

語法筆記：

處理並儲存藍牙資料

資料印出來之後，步數資料是每偵測到一次，dataString 會收到 "1"，需要有一個計步數的變數 stepCount 來記錄偵測到的總步數。另外由印出來的溫度資料得知，此是可以直接儲存在顯示在畫面上。

步驟D 程式碼

在 @Published 變數宣告區 宣告兩個變數，其中一個為 @Published 變數：

```
@Published var dataReceived = [" ", " "]
var stepCount = 0
```

在要儲存步數與溫度的 dataString 資料，我們使用一個字串陣列 dataReceived 來儲存，並具有@Published 修飾符。dataReceived 的資料將會在 App 畫面呈現。stepCount 這個變數是用來計算目前的總步數。

修改 C3 函數

在 class BLEManager 內，修改 peripheral(_:didUpdateValueFor:error:) 這一個函數，加入下方兩個紅色方框內的程式碼。

```
func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor characteristic:
 CBCharacteristic, error: Error?) {
    if let data = characteristic.value {
        let dataString = data.map{ String(format: "%d", $0) }.joined()
        if characteristic.uuid == CBUUID(string: stepCharID) {
            print("Step data: " + dataString)
            if dataString == "1" {
                stepCount += 1
                dataReceived[0] = String(stepCount)
            }
        } else if characteristic.uuid == CBUUID(string: tempCharID) {
            print("Temperature data: " + dataString)
            if dataString != dataReceived[1] {
                dataReceived[1] = dataString
            }
        }
    }
}
```

① 計算並儲存總步數：符合 stepCharID 的資料，每次 dataString 收到 1 的值時，讓 stepCount 加 1 。stepCount 為整數型別，需要使用String(stepCount) 轉成文字後才能儲存至 dataReceived[0]。

② 當溫度值有變化時，儲存新資料到 dataReceived[1] 。

畫面顯示藍牙接收資料

在藍芽管理器 BLEManager 內，有 @Published 特性的變數 dataReceived 可以被 ContentView 的 @ObservedObject 觀察到其變化。

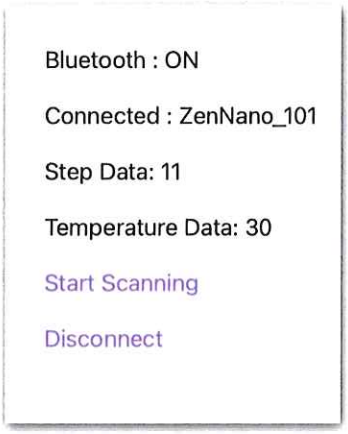
步驟 E 程式碼

在 ContentView 的 body 內，在 App 畫面上顯示目前的步數資料與溫度資料：

```
Text("Step Data: \(bleManager.dataReceived[0])")
Text("Temperature Data: \(bleManager.dataReceived[1])")
```

測試連線，看看 central App 是否正確收到資料？

Connect



Disconnect

目前斷開連線之後，畫面的資料不會清空，重新連線之後，步數會繼續被計算，若要再斷開連線後 dataReceived 的資料要被清空，要從 B6 的函數來修正。

找到 B6 函數 centralManager(_:didDisconnectPeripheral:error:)，將 dataReceived 修改為空字串陣列，stepCount 要歸零，如下：

```
func centralManager(_ central: CBCentralManager, didDisconnectPeripheral
 peripheral: CBPeripheral, error: Error?) {
    isConnected = false
    dataReceived = [" ", " "]
    stepCount = 0
}
```

測試看看斷開連線後，App 上的資料是否被清空？

章節筆記

語法疑問：

教學重點：

反思回饋：

MQTT 基本觀念

MQTT (Message Queue Telemetry Transport) 是一種輕量級的通訊傳輸協定，設計用於物聯網 (Internet of Things, IoT) 的環境中，適用於低頻寬、有限資源的網路環境。 MQTT 的運作方式是基於訂閱/發布模式，透過代理者（Broker）來管理所有客戶端的連接和訊息交換。

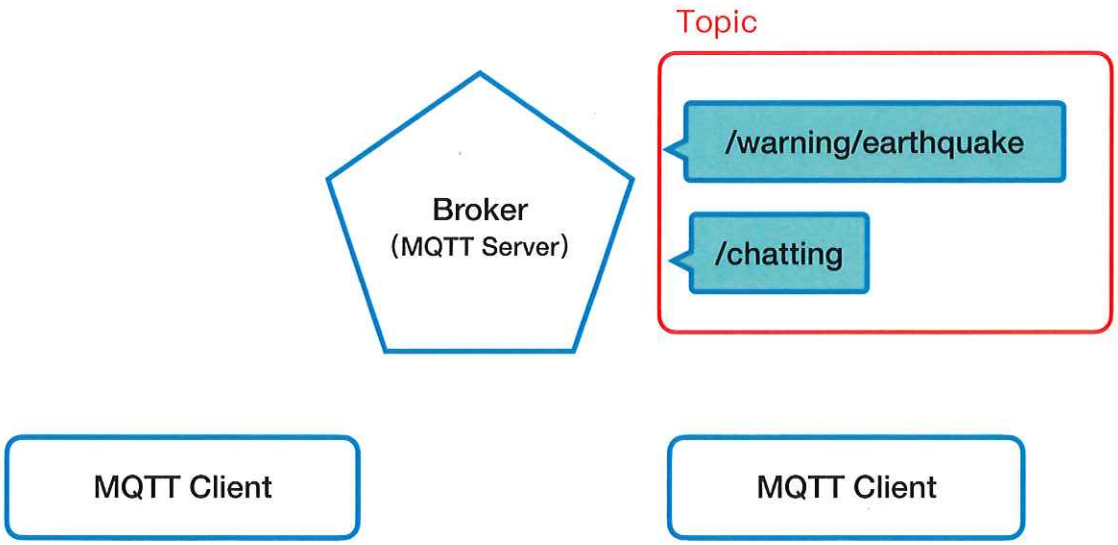
MQTT 的歷史可以追溯到1999年，當時由 IBM 的 Andy Stanford-Clark 和 Arlen Nipper 共同創造出來。他們的目標是為了滿足石油和天然氣管道的監控需求，這需要一種輕量、可靠且有節能效果的通訊協定。這也正是 MQTT 後來在物聯網應用中得以廣泛使用的原因。

Broker (MQTT Server) and Topic

MQTT 的訊息傳遞是基於訂閱 (Subscribe) /發佈 (Publish) 模式，來傳遞消息給MQTT使用者 (MQTT Client)。

所有訊息都會經過一個代理者（Broker）也就是伺服器（MQTT Server）做為這些訊息的中介者，Broker 會負責管理與發送消息，是這些訊息傳輸的中心。Broker 也負責處理所有的MQTT連接、訂閱和取消訂閱的請求，並確保訊息安全可靠地傳遞給對應的接收者。

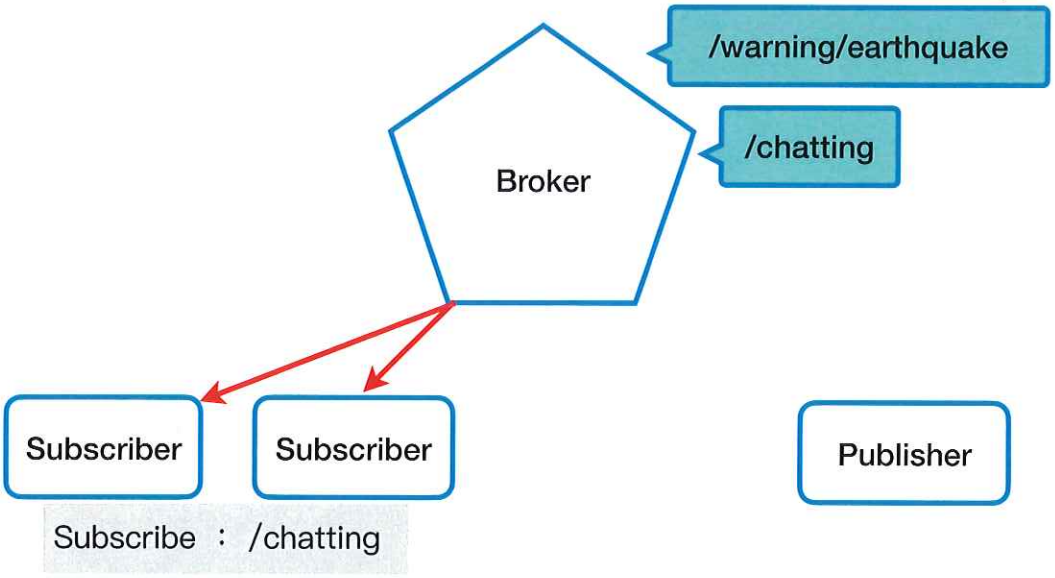
Broker 在管理訊息的時候，使用主題 Topic 來分類管理訊息，一個 Broker 底下可以有多個 Topic，例如：/chatting 、/warning/earthquake ，如下圖：



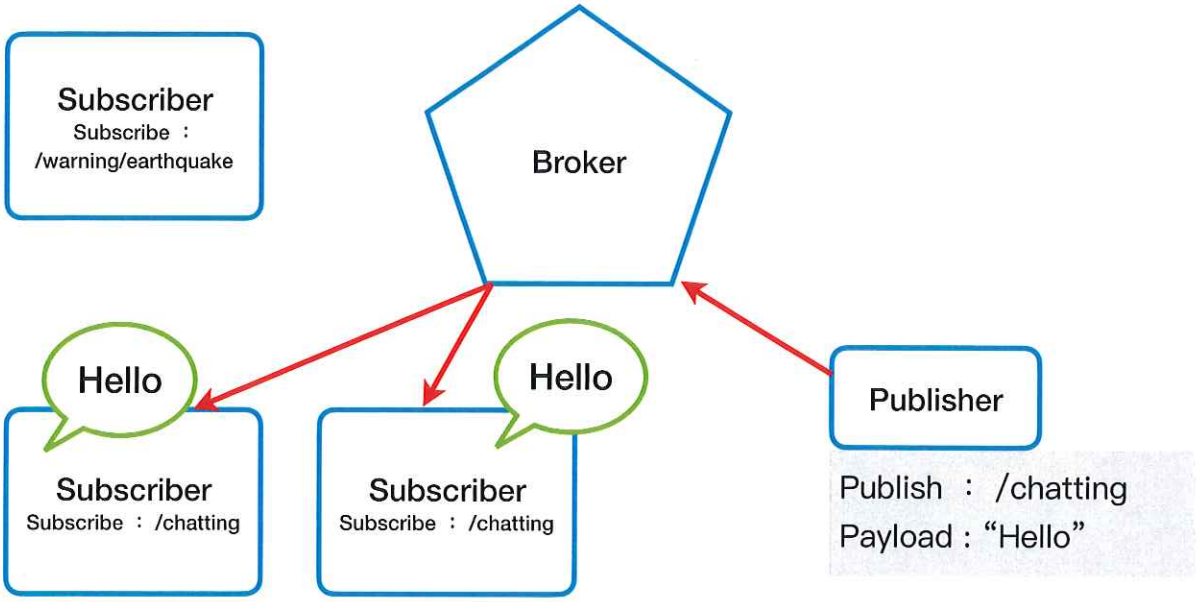
Subscriber and Publisher (MQTT Client)

MQTT 使用主題 (Topic) 作為訊息管理的機制。

用戶端可以訂閱感興趣的主題，如下圖灰色方框 Subscribe ： /chatting，當有新的訊息發布至該主題時，Broker 會將訊息發送至所有訂閱該主題的訂閱者 (Subscriber)，一個主題可以有多個訂閱者，如下圖：



MQTT 訊息的發布者（Publisher）會將特定的訊息發布到特定的主題，如下圖灰色方框。Broker 會接受這個訊息，並發送訊息給該主題的訂閱者，如下圖：



一個裝置可以只是訂閱者的角色或只有發布者的角色，但也有可能同時是訂閱者與發佈者。這取決於你的裝置的功能與需求來設定。

MQTT Quality of Service (QoS)

Quality of Service (QoS) 為 MQTT 的訊息傳遞方式設定，分為三等級：

QoS 0：訊息至多傳送一次，無法保證是否接收。

QoS 1：訊息至少傳送一次，可能接收多次，可能導致重複。

QoS 2：訊息確保僅傳送並接收一次，傳遞完整可靠，但最耗時和影響效能。

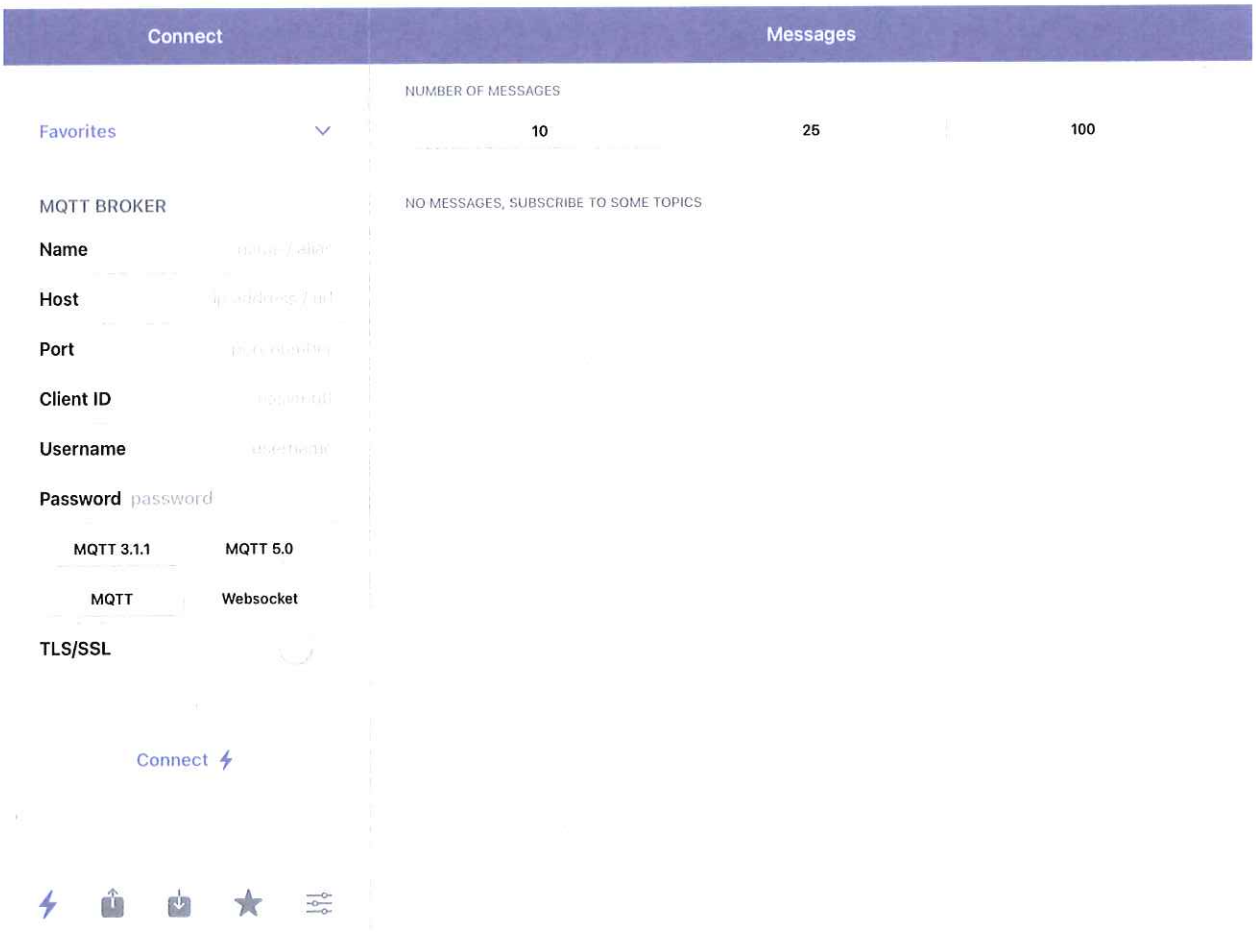
選擇哪種 QoS 等級取決於你的應用的特定需求，例如訊息傳遞的重要性、系統的頻寬限制和設備的能力等。

EasyMQTT (App)

EasyMQTT 是一款跨平台的 MQTT 客戶端工具，可在 Mac，iPhone 和 iPad 上使用。它提供用戶簡單容易操作的介面，讓使用者可以在各種裝置上輕鬆的使用 MQTT 通訊協定與 Broker 連接，發布訊息和訂閱主題接收訊息。

EasyMQTT

掃描左方QR Code 下載並開啟 EasyMQTT App ：



連線 MQTT Broker

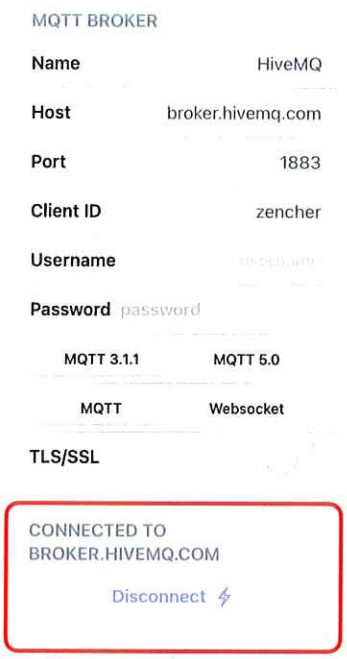
這次課程中，我們使用 HiveMQ 提供的的免費 MQTT Broker： broker.hivemq.com ，此網址允許任何 MQTT 用戶端進行連線，發布和訂閱消息。

連線 MQTT Broker 設定步驟：

1. Name：為這個連線設定的儲存名字，可以自行設定。
2. Host：連線的 Broker 網路位置(url) 或 ip 。
3. port：標準的未加密連線通常使用 port 1883。
4. Client ID：請自行選擇定義 Client ID。

注意：在同一個 Broker 上，每個使用者的 Client ID 不能和別人相同，否則可能會產生衝突。

5. Username and Password：依照連線的 Broker 規範，本次課程中 broker.hivemq.com 連線時，Username 與 Password 空白即可。
6. 按下 Connect 鍵後，若成功連線，會如右圖所示。

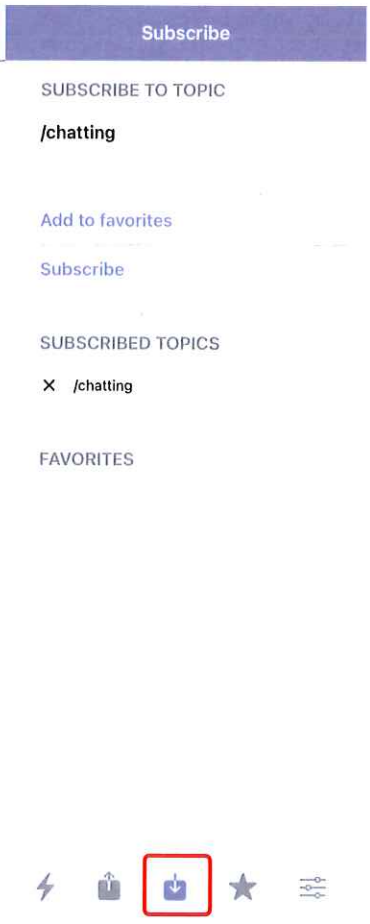


Subscribe Topic ：/chatting

在左側連線區塊下方，點選接收訊息的圖示。

進入 Subscribe 設定：

1. 在第一個欄位輸入：/chatting
2. 點選 Subscribe 訂閱 Topic
3. 成功訂閱時會在 Subscribed Topics 看到訂閱的主題。

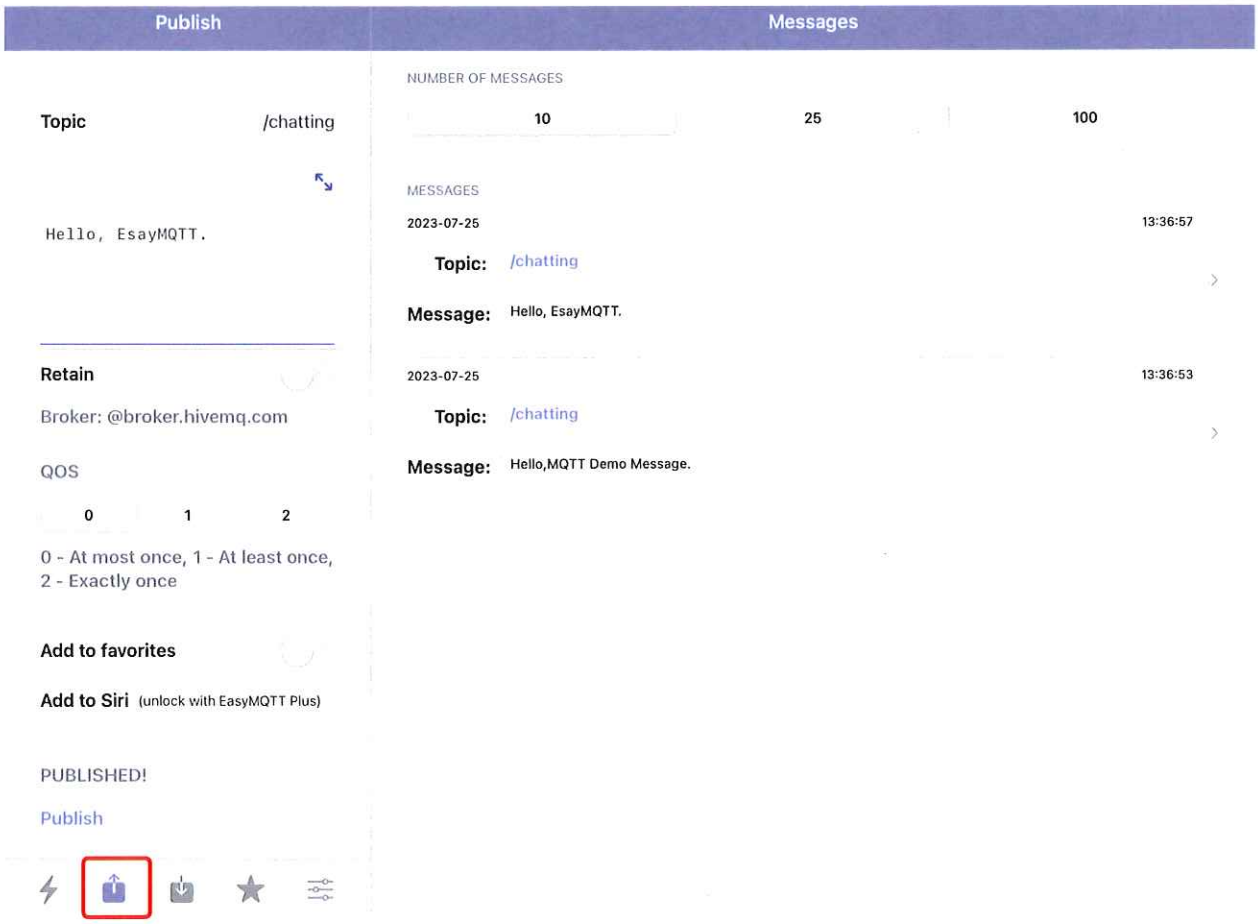


Publish

成功訂閱主題後，應該會收到目前存在 Broker 的 retain (保留)訊息。

在左側連線區塊下方，點選發送訊息圖示，進入發送訊息介面：

1. 輸入要對哪一個 Topic 發送訊息：/chatting。
2. 輸入訊息內容： "Hello, EsayMQTT."
3. 按下 Publish ，訊息就會被傳至連線的 Broker。
4. 在右方 messages 區域，因為前一個步驟我們訂閱了/chatting 這個主題，所以有任何人在這個 Broker 的 /chatting 發送訊息，都會在 messages 區域被接收。



在訊息發送時，retain 選項若設定為 true 時，Broker 會存儲此訊息為這個主題的最後一條消息，新訂閱用戶之後的連線上 Broker ，可以立即接收到此最新的消息。

試試看你的訊息是否成功 Publish ，並接收 Broker 到其他人發送的訊息？

章節筆記

語法疑問：

教學重點：

反思回饋：

MQTT Client App

在實作 MQTT Client App 的專案中，我們將使用兩個不同的 MQTT 用戶端套件，練習使用 closure 或是 delegate 來開發 MQTT Client App。

- LightMQTT ：此 MQTT 套件使用 Closure 語法來處理 MQTT 的連線與訊息接收。

注意：LightMQTT 只能在 iPad 的 Swift Playgrounds 使用。

- SwiftMQTT ：這是一個在 GitHub 的公開 Swift MQTT Client 套件，使用 delegate 模式來處理訊息接收。SwiftMQTT 可以在 iPad 與 Mac 的 Swift Playgrounds 使用。

注意：目前在 Swift Playgrounds 上使用 MQTT 套件可能會遇到連線不穩定的狀況，在 Mac 上使用 Xcode 套件 CocoaMQTT（https://github.com/emqx/CocoaMQTT.git） 可以解決連線不穩定的問題，。

Closure 語法處理 MQTT 訊息 – LightMQTT

此專案由 MQTTLight_Template 開始，其中已準備好 LightMQTT.swift 這個套件。
這個專案將分為 A ~ C 步驟完成：

步驟A. 在畫面 VStack{ } 的.onAppear { } 內建立 MQTT 連線，並訂閱 Topic。

步驟B. 新增一個按鈕與輸入欄位來發布訊息。

步驟C. 在.onAppear { } 內執行接收訊息，將訊息儲存為一個 Array後，在 App 畫面中將接收到的訊息使用 List 陳列出來。

```
import SwiftUI

struct ContentView: View {
    //宣告變數區

    var body: some View {
        VStack {
            //C2.將接收的訊息顯示在畫面上


            //B.發布訊息


        }.onAppear {
            //A.MQTT 連線與 Subscribe Topic

            //C1.接收訊息

        }
    }
}
```

步驟A 程式碼

在步驟A中，我們要建立 MQTT 連線，並訂閱 Topic ，分為 A1~A2 兩部分。

A1. 建立MQTT 連線

在 宣告變數區 宣告一個 client ，型別為 LightMQTT，並使用強制解開符號！，表示當 client 被使用時，我們確保它會有值。

```
struct ContentView: View {
    //宣告變數區
    @State var client:LightMQTT!
```

接著在 .onAppear{ } 內，宣告一個 options 為 MQTT 的連線選項設定，並設定 client 的內容，如下：

```
.onAppear {
    //A.MQTT 連線與 Subscribe Topic
    let options = LightMQTT.Options(port: 1883, clientId: "zencher")
    client = LightMQTT(host: "broker.hivemq.com", options: options)
```

options 為 MQTT 連線設定 LightMQTT.Options(port: 1883, clientId: "zencher")，其中參數 port 設定未加密連線連接口為 1883，使用者 ID 為 "zencher" 。

client 為 LightMQTT()的實例，連線至 host: "broker.hivemq.com" ，連線選項為已設定好的 options 。

注意：每個使用者的 Client ID 不能和其他人相同。

語法筆記：

A2. 訂閱 Topic

在 宣告變數區 宣告一個常數 subTopic 儲存訂閱的 Topic 為 "/chatting" ：

```
struct ContentView: View {
    //宣告變數區
    @State var client:LightMQTT!
    let subTopic = "/chatting"
```

在 .onAppear{ } 內，設定訂閱 Topic 。

```
.onAppear {
    //A.MQTT 連線與 Subscribe Topic
    let options = LightMQTT.Options(port: 1883, clientId: "zencher")
    client = LightMQTT(host: "broker.hivemq.com", options: options)

    client.connect { success in
        if success {
            client.subscribe(to: subTopic)
            print("Connect Successful. Subscribed to \(subTopic).")
        } else {
            print("Connect Failed.")
        }
    }
```

執行連線 client.connect，這個 closure 會傳出一個 Bool 值 success，為連線成功與否。

如果成功連線 if success { } ，使用 .subscribe(to: subTopic) 訂閱 Topic 並在主控台印出連線成功訊息，若連線失敗，在主控台印出連線失敗 "Connect Failed."。

語法筆記：

步驟B 程式碼

新增一個按鈕與輸入欄位來發布訊息。

在 宣告變數區 宣告一個 inputMessage 來儲存輸入欄位的內容：

```
@State var inputMessage = ""
```

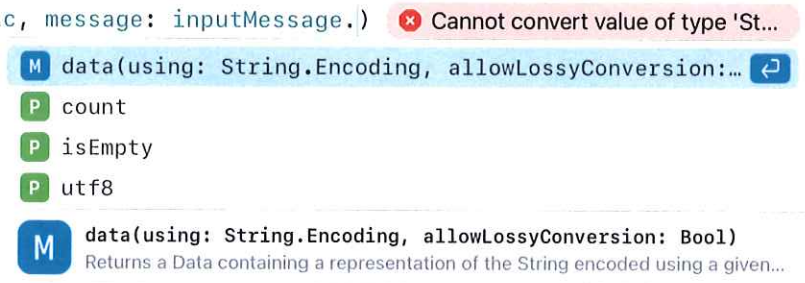
在 ContentView 的 VStack { } 內，

新增一個輸入欄位 TextField，使用 $ 賦予 TextField 修改 inputMessage 的權力：

```
//B.發布訊息
TextField("Publish Message", text: $inputMessage)
    .padding()
    .textFieldStyle(RoundedBorderTextFieldStyle())
```

並新增一個按鈕 Button 來發送輸入的訊息 inputMessage：

```
Button("Send"){
    client.publish(to: subTopic, message: inputMessage.)
}
```



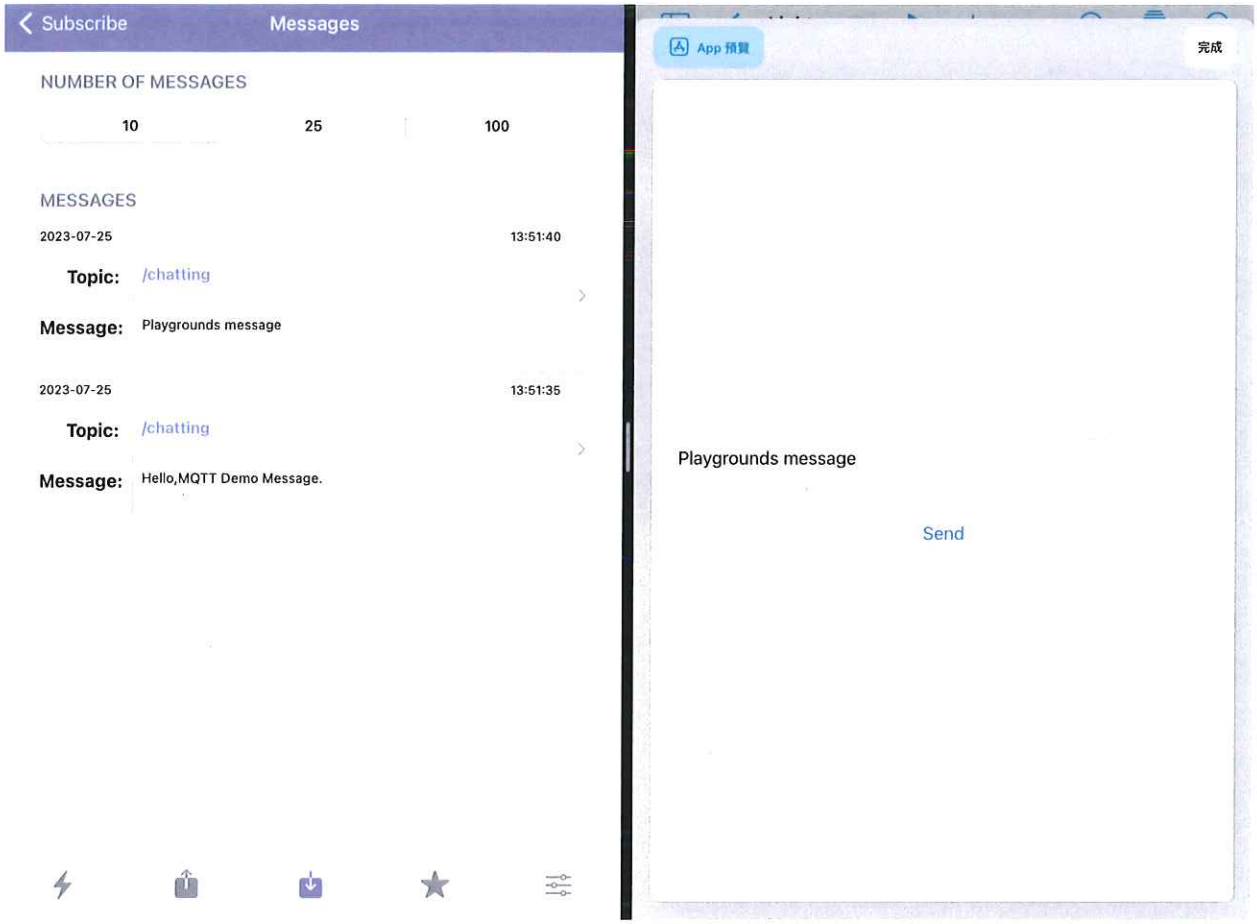
按鈕內執行 client 的 .publish 的方法發布訊息，其中參數 to: 為對哪一個 Topic 發送訊息，message: 為訊息內容。

inputMessage 要使用 .data 的方法轉換格式，由 String 轉為Data 資料。以 UTF-8 的編碼方式來將 String 轉換成 Data。allowLossyConversion: true 這部分是告訴 Swift 當某些字元無法用 UTF-8 編碼時，是否允許丟棄這些字元以完成轉換。

完成的 Button 按鈕如下：

```
Button("Send"){
    client.publish(to: subTopic, message: inputMessage.data(using: .utf8,
     allowLossyConversion: true))
}
```

完成步驟B後，在 App 畫面應該可以成功發出訊息：



注意：EasyMQTT 無法在背景執行接收訊息，需要將 EasyMQTT 與 Playgrounds 這兩個 App 使用並行視窗，在 EasyMQTT 就可以看到發送訊息的結果。

語法筆記：

步驟C 程式碼

在 宣告變數區 宣告一個陣列變數來儲存接收的訊息：

```
@State var receivedMessages: [String] = []
```

C1 程式碼：設定接收訊息

在 .onAppear{ } 內的 **C1** 區域，使用 client 的 .receivingMessage 的方法接收訊息：

```
client.receivingMessage = { topic, message in
    receivedMessages.append(message)
}
```

receivingMessage 為一個 closure，若接收到訊息時，會傳出兩個回傳值，一個目前已訂閱的 topic ，另一個是在這個 topic 下接收到的訊息 message 。

receivedMessages.append(message) 會將接收到的訊息，新增至 receivedMessages 的陣列內。

C2 程式碼：將接收訊息使用 **List** 呈現在 **App** 畫面上

在 ContentView 的 VStack { } 內的 **C2** 區域，使用 List 列出所有接收到的訊息：

```
VStack {
    //C2.將接收的訊息顯示在畫面上
    List(receivedMessages, id:\.self){ message in
        Text(message)
    }
```

List 使用 closure 的寫法，將 receivedMessages 每一個元素使用列表呈現，id: \.self 提供了每個元素在列表中的唯一標識。

在 List 的 closure 內使用 message 這個變數拿到每一個 receivedMessages 的元素，並使用 Text(message) 在列表內呈現所有訊息。

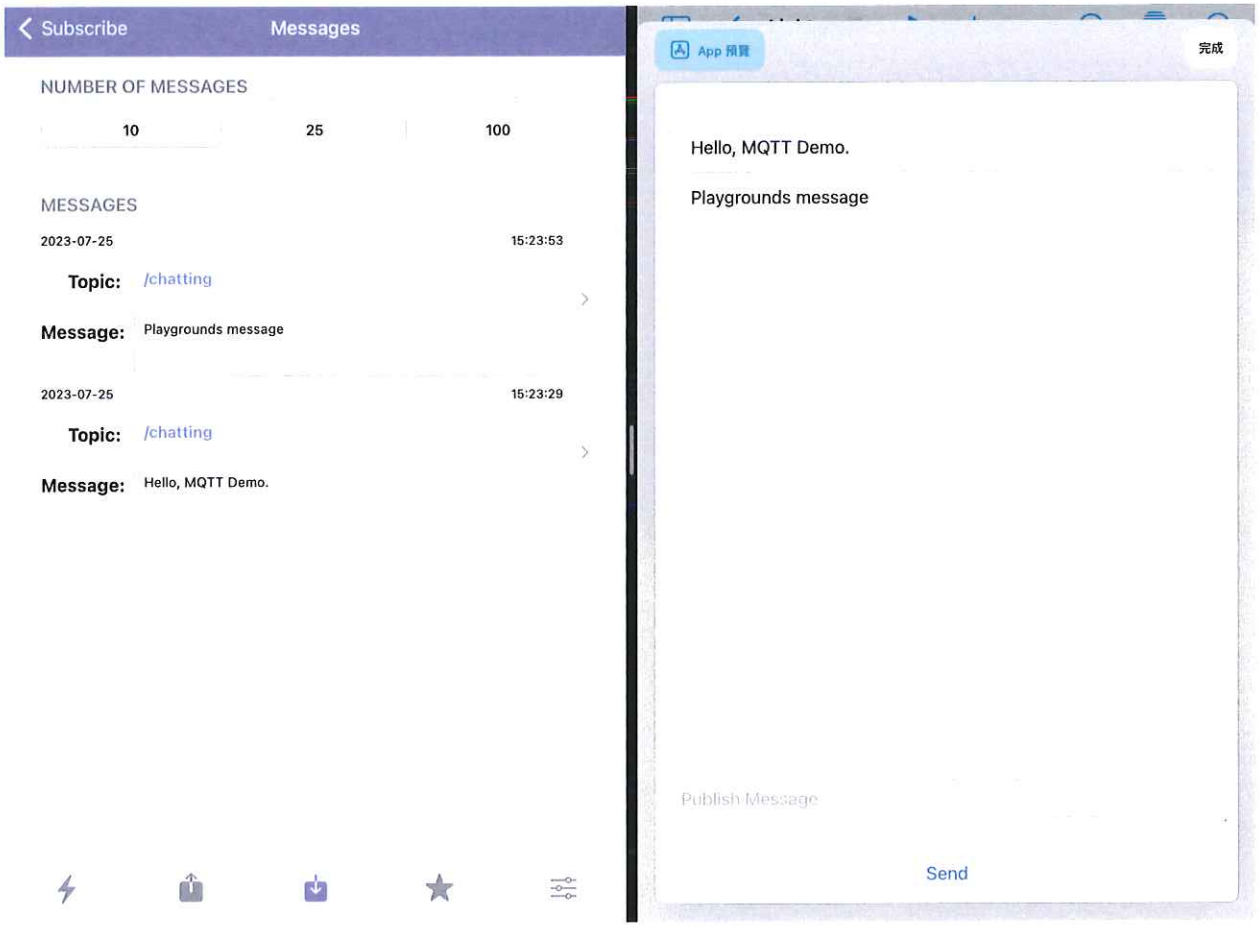
最後如果希望送出訊息後，訊息欄會被清空，要在步驟B的按鈕內加入：

```
Button("Send"){
    client.publish(to: subTopic, message: inputMessage.data(using: .utf8,
     allowLossyConversion: true))
    inputMessage = ""
}
```

注意：這行一定要加在發送訊息之後。

測試看看現在的App 是否能成功收發訊息了？

注意：有可能會遇到連線不穩定，或是收發訊息不穩定的情況。



章節筆記

語法疑問：

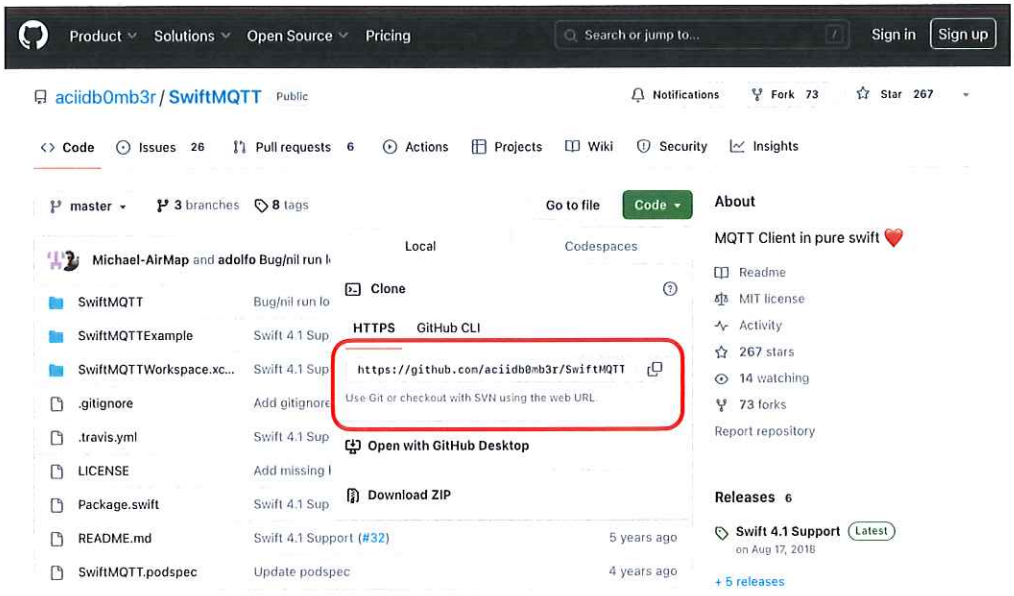
教學重點：

反思回饋：

Delegate 模式處理 MQTT 訊息 – SwiftMQTT

SwiftMQTT 是一個在 GitHub 上的公開 Swift MQTT Client 套件，我們將使用此套件，練習使用 delegate 模式來完成另一個 MQTT Client App。

將 SwiftMQTT 第三方套件載入 Playgrounds 的方法如下：
從網頁獲取 SwiftMQTT 套件的 URL ：https://github.com/aciidb0mb3r/SwiftMQTT.git



SwiftMQTT

在 App 專案中加入 Swift 套件，貼上套件 URL 後，點選加入到 App Playground ：

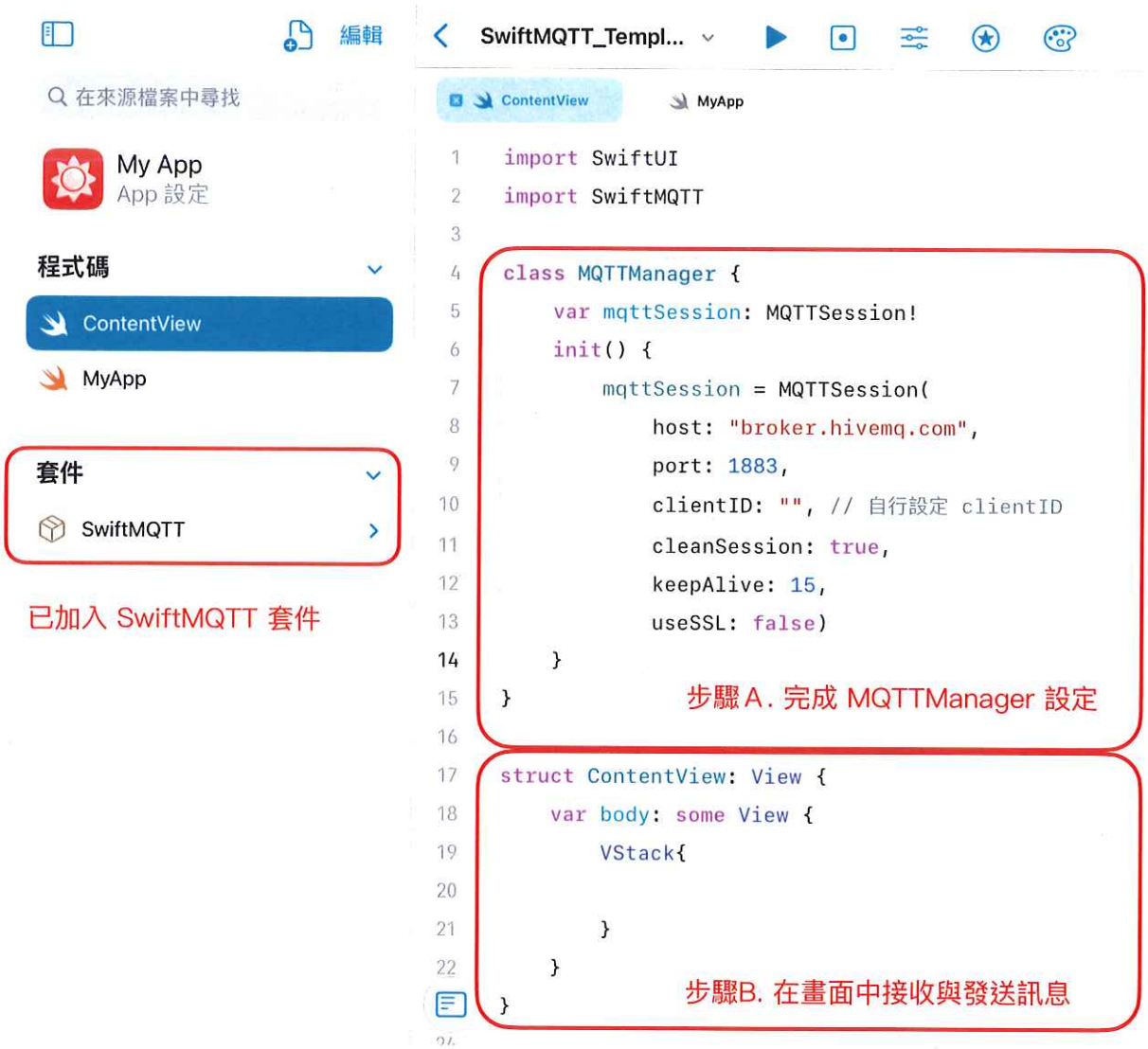
Mac 版的 Playgrounds 在右上角的檔案內可以加入套件。檔案>>加入套件.....

這個專案將從 SwiftMQTT_Template 這個檔案開始，已將下列事項準備完成：

1. 加入第三方套件 SwiftMQTT，並 import SwiftMQTT 至 swift 檔案內。
2. 建立一個 MQTTManager 為 MQTT 管理器。其中由 mqttSession 負責處理 MQTT 連線與訊息收發，使用 init{ } 初始化 mqttSession 並完成連線 broker 的參數設定。

注意：仍需自行設定 clientID

由 SwiftMQTT_Template 開始，完成此專案可分為兩大部分，分別為 步驟 A 與 B：



步驟A. 完成 MQTTManager 設定：

將 MQTTManager Class 設定為一個可觀察物件，並遵循 MQTTSessionDelegate ，將 mqttSession 代理 delegate 設為 MQTTManager。完成 mqttSession 初始化 init{ } 的內容，並在 MQTTManager 內設定函數處理連線與訊息收發。

步驟 B. 在 App 畫面中接收與發送訊息：

在 ContentView 中使用修飾符 @ObservedObject ，宣告一個被觀察的物件 mqttManager ，其為 MQTTManager() 的實例。在畫面上，將接收的訊息使用 List 列出來，並加入一個 TextField 與 Button 處理訊息發布。

步驟 A 程式碼

完成 MQTTManager 相關設定，可以再細分為 A1 ~ A5 部分：

```
class MQTTManager {
    var mqttSession: MQTTSession!
    //A2. 宣告一個常數儲存要訂閱的 Topic，與一個 @Published 變數，儲存接收到的訊息


    //A3. 在 init() 內設定 mqttSession 的 delegate，並連線至 Broker 與訂閱 Topic
    init() {
        mqttSession = MQTTSession(
            host: "broker.hivemq.com",
            port: 1883,
            clientID: "", // 自行設定 clientID
            cleanSession: true,
            keepAlive: 15,
            useSSL: false
        )
    }
    //A4. 設定 4 個函數，處理 MQTT 收發訊息與連線偵測

}
```

語法筆記：

A1 程式碼

將 MQTTManager 設為可觀察物件 ObservableObject 並遵循 MQTTSessionDelegate：

```
4   class MQTTManager: ObservableObject, MQTTSessionDelegate {
```

Type 'MQTTManager' does not conform to protocol 'MQTTSessionDelegate'

設定完成後會出現一個錯誤。目前 MQTTManager 並沒有遵循 MQTTSessionDelegate 的協議，要修正此錯誤須先在 A4 程式碼 區域加入三個函數來解決此錯誤。

使用選單加入這三個函數，讓 MQTTManager 遵循 MQTTSessionDelegate 的協議：

```
    //A4. 設定 4 個函數，處理 MQTT 收發訊息與連線偵測
    m
```

```
M  mqttDidAcknowledgePing(from: MQTTSession)
M  mqttDidReceive(message: MQTTMessage, from: MQTTSessio…
M  mqttDidDisconnect(session: MQTTSession, error: MQTTSe…
K  mutating
```

```
M  mqttDidAcknowledgePing(from: MQTTSession)
   沒有可用文件。
```

依順序加入函數後，這個錯誤會消失。目前函數的內容為空的，之後會再回來設定內容。

```
//A4. 設定 4 個函數，處理 MQTT 收發訊息與連線偵測
func mqttDidReceive(message: MQTTMessage, from session: MQTTSession) {

}
func mqttDidAcknowledgePing(from session: MQTTSession) {

}
func mqttDidDisconnect(session: MQTTSession, error: MQTTSessionError) {

}
```

A2 程式碼

宣告一個常數 subTopic 儲存要訂閱的 Topic，為一個 String。另外宣告一個 @Published 變數 receivedMessages ，為一個空的 String 陣列，用來儲存接收到的訊息。

```
//A2. 宣告一個常數儲存要訂閱的 Topic，與一個 @Published 變數，儲存接收到的訊息
var subTopic = "/chatting"
@Published var receivedMessages: [String] = []
```

A3 程式碼

在 init() 內設定 mqttSession 的 delegate 為 self，self 為 MQTTManager 這個 class：

```
mqttSession.delegate = self
```

mqttSession 使用 .connect 連線至 Broker ，使用選單加入 connect 方法。

```
mqttSession.connect(completion: MQTTSessionCompletionBlock?)
```

此為一個 closure，點選藍色區塊，在鍵盤上按下 return (enter) 鍵，playgrounds 會自動加入 closure 語法，其會傳出一個參數 error。

```
mqttSession.connect { error in
    code
}
```

第一個紅色方框：處理 connect error

使用 guard 語法處理，若連線成功 error == .none 為 true 則不會執行 else { } 內容，error == .none 為 false 執行 else { } 內容，在主控台印出 "Connect Error: \(error)" 。在此套件中 .none 代表沒有 error 。

```
mqttSession.delegate = self
mqttSession.connect { error in
    guard error == .none else {
        print("Connect Error:\(error)")
        return
    }
    self.mqttSession.subscribe(to: self.subTopic, delivering: .atLeastOnce,
     completion: nil)
}
```

處理 connect error

訂閱 Topic

第二個紅色方框：訂閱 Topic

成功連線後，mqttSession 使用 .subscribe 的方法來訂閱 Topic 。

.subscribe 的參數： to: 設定訂閱 Topic 為 self.subTopic ，delivering: .atLeastOnce 設定資料傳輸服務品質 QoS 為至少傳送一次，completion: nil 代表完成後不做其他任務 。

A4 程式碼

設定四個函數的內容，如下表：

設定順序	呼叫	函數名	功能說明
A4.1	主動	mqttPublish(publishMessage:)	執行 publish 發布訊息
A4.2	被動	mqttDidReceive(message:from:)	接收到訊息後儲存於 receivedMessages
A4.3	被動	mqttDidAcknowledgePing(from:)	偵測 MQTT 是否處於連線狀態
A4.4	被動	mqttDidDisconnect(session:error:)	MQTT 斷線時，在主控台印出訊息

A4.1 函數 mqttPublish(publishMessage :)

設定一個函數 mqttPublish 來發布訊息，其中包含了一個傳入參數 publishMessage，型別為 String ：

```
func mqttPublish(publishMessage: String) {
    if let data = publishMessage.data(using: .utf8, allowLossyConversion: true) {

    }
}
```

將 publishMessage 使用 .data 的方法，將 String 轉為 Data 資料，並以 UTF-8 的編碼方式來將 String 轉換成 Data。allowLossyConversion: true 這部分是告訴 Swift 當某些字元無法用 UTF-8 編碼時，是否允許丟棄這些字元以完成轉換。

用 if let 確保 data 完成轉檔後，再使用 mqttSession 的 .publish 方法發布 data 訊息。

```
func mqttPublish(publishMessage: String) {
    if let data = publishMessage.data(using: .utf8, allowLossyConversion: true) {
        mqttSession.publish(data, in: subTopic, delivering: .atLeastOnce, retain: false,
        completion: nil)
    }
}
```

其中 .publish 參數設定：

資料 data 在 in: subTopic 這個主題 Topic 發送。

QoS 設定為至少發送一次 delivering: .atLeastOnce。

retain 設定為 false，代表此訊息發送給目前連線的所有訂閱者後，Broker 就會丟棄該消息。也就是說，新來的訂閱者將不會收到舊的消息，只會收到自己訂閱之後發布的消息。

completion 設定為 nil，這代表著當 publish 完成時，沒有任何操作會被執行。

A4.2 函數 mqttDidReceive(message :from:)

當接收到訊息時，要先對訊息處理進行轉檔。

```
func mqttDidReceive(message: MQTTMessage, from session: MQTTSession) {
    if let messageString = String(data: message.payload, encoding: .utf8) {
        receivedMessages.append(messageString)
    }
}
```

將收到的訊息 message.payload ，使用 .utf8 的解碼方式將資料由 Data 轉為 String。並用 if let 確定轉檔成功後，將 messageString 新增至 receivedMessages 陣列中。

A4.3 函數 mqttDidAcknowledgePing(from:)

偵測 MQTT 是否處於連線狀態，為了維護與 MQTT 的連線，用戶端定期向 Broker 發送 ping 請求，如果 Broker 在設定的時間內沒有回應 ping 請求，用戶端會認定連線已中斷。

```
func mqttDidAcknowledgePing(from session: MQTTSession) {
    print("acked")
}
```

若接收到 Broker 的回應後，在主控台印出 "acked" ，acked 是 acknowledged"的縮寫，代表 "已確認" 的意思。

A4.4 函數 mqttDidDisconnect(session:error:)

若偵測到與 MQTT Broker 的連線已中斷時，在主控台印出訊息斷線訊息：

```
func mqttDidDisconnect(session: MQTTSession, error: MQTTSessionError) {
    print("Disconnected")
}
```

語法筆記：

步驟B 程式碼

在這個步驟要完成 ContentView 的畫面。

先在 ContentView宣告區域 宣告變數：

```
@ObservedObject var mqttManager = MQTTManager()
@State var inputMessage = ""
```

用 @ObservedObject 修飾符宣告變數 mqttManager 為一個 MQTTManager() 實例，並使用 @State 宣告一個 inputMessage 的空字串，來儲存要發送的訊息。

接下來的步驟分成 B1 ~ B2：

```
struct ContentView: View {
    @ObservedObject var mqttManager = MQTTManager()
    @State var inputMessage = ""

    var body: some View {
        VStack{
            //B1.將接收訊息使用 List 呈現

            //B2.新增輸入欄位與按鈕來發布訊息

        }
    }
}
```

B1 程式碼：將接收訊息使用 **List** 呈現在 **App** 畫面上

在 ContentView 的 VStack { } 內的 B1 區域，使用 List 列出所有接收到的訊息：

List 使用 closure 的寫法，將 mqttManager.receivedMessages 每一個元素使用列表呈現，id: \.self 提供了每個元素在列表中的唯一標識。

```
List(mqttManager.receivedMessages, id: \.self) { message in
    Text(message)
}
```

在 List 的 closure 內使用 message 拿到每一個 mqttManager.receivedMessages 的元素，並使用 Text(message) 在列表內呈現所有訊息。

B2 程式碼：將接收訊息使用 **List** 呈現在 **App** 畫面上

加入輸入欄位，輸入的內容將存在 inputMessage ，並使用 $ 賦予 TextField 修改 inputMessage 的權力：

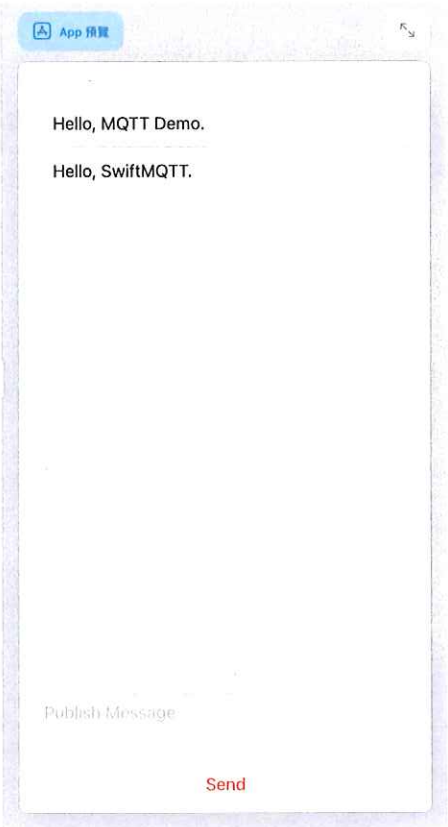
```
TextField("Publish Message", text: $inputMessage)
    .textFieldStyle(RoundedBorderTextFieldStyle())
```

加入一個按鈕，讓 mqttManager 執行 .mqttPublish 來發送輸入的訊息 inputMessage，並且發送完後將輸入欄位清空：

```
Button("Send"){
    mqttManager.mqttPublish(publishMessage: inputMessage)
    inputMessage = ""
}
```

測試看看這個 SwiftMQTT App 是否能成功收發訊息？

注意：有可能會遇到連線不穩定，或是收發訊息不穩定的情況。



章節筆記

語法疑問：

教學重點：

反思回饋：

MQTT Client App Challenge

* 在每一個檔案中，都需要確認 MQTT 連線 clientID
(不能與其他人相同，建議使用英文與數字組合，否則有可能出現連線錯誤)

SwiftMQTT_Light – IoT 應用

Challenge A

目前有一個模擬 IoT 應用的 Light App，訂閱了 /lighting 這個 Topic ，接收訊息來控制燈開關，訊息必須為以下的 Json 格式，才能對 App 內的燈進行開或關的切換。

複製一個你的 SwiftMQTT App ，修改程式碼完成此任務！

開燈 Json 訊息	關燈 Json 訊息
{ "isOn":1 }	{ "isOn":0 }

注意：Json 訊息中 1 與 0 為 Int 型別

SwiftMQTT_ChatRoom

Challenge B

建立一個 MQTT 即時聊天室，訂閱了 /chatRoom 這個 Topic。複製一個你的 SwiftMQTT App 修改程式碼完成此任務！

SwiftMQTT_ChatRoomPlus

Challenge C

加入可以隨時修改使用者暱稱的功能。

Challenge D

讓聊天室輸入欄位可以按下 enter 後直接送出訊息。

Challenge X

想想這個即時聊天室功能都完整了嗎？ 還有什麼想要加入的功能呢？
如何美化聊天室的畫面？

Challenge A 步驟提示

1. 修改訂閱 Topic 為 /lighting
2. 修改 func mqttPublish ：
   (1) 新增 Int 變數 openLight，判斷輸入字串，來決定 openLight 為 0 或 1。
   (2) 宣告一個 [key: value] 的常數為 publishMessage = ["isOn" : openLight] 。
   (3) 將 publishMessage 使用 JSONSerialization 轉為 json 格式的 data ， 並將訊息發送出去，data 訊息 publish 方式與之前相同。

Challenge B 步驟提示

1. 定義一個名為 ChatJson 的 JSON Model ，裡面包含兩個 key: value pairs ，儲存使用者名稱與使訊息內容。此 JSON Mode 用來發送與解析 JSON 格式的訊息資料。
2. 修改訂閱 Topic 為 /chatRoom ，並宣告一個新變數 userName 為聊天室使用者暱稱。
3. 修改變數 receivedMessages 型別，為一個 ChatJson 陣列。
4. 修改 mqttPublish(publishMessage:) ：
   (1) 宣告一個名為chatMessage 的 ChatJson 物件，儲存 userID 與輸入訊息 publishMessage。
   (2) 用 JSONEncoder 將 chatMessage 轉為 Json 格式的 data，並將訊息發送出去，data 訊息 publish 方式與之前相同。
5. 修改 mqttDidReceive(message:from:) 函數，使用 JSONDecoder() 將訊息 message.payload 解碼為 ChatJson 物件，並將解碼完成的 ChatJson 物件新增至 receivedMessages 。
6. 修改 List 呈現方式，判斷訊息的發送者，將自己發送的訊息靠右，他人發送的訊息靠左。

作者：Michael Pan