VIETNAMESE-GERMAN UNIVERSITY
KARLSRUHE UNIVERSITY OF APPLIED SCIENCES

MASTER THESIS

DEVELOPING MQTT DASHBOARD WITH SCALA AND AKKA

Author: Tan Dung, Tran
VGU Student ID: 18859
HKA Matriculation Number: 81182

Referee: Prof. Dr. Thorsten Leize
Co-referee: Prof. Dr. Thomas Westermann
Supervisor: Prof. Dr. Thorsten Leize

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in Faculty of

Mechatronics and Sensor Systems Technology

2022

Contents

ACKNOWLEDMENT

I would like to express my very great appreciation to Prof. Dr. Thorsten Leize, my thesis supervisor, for his valuable and constructive suggestions during the planning and development of this thesis. His willingness to give his time so generously has been very much appreciated.

Besides, I also want to thank my second marker Prof. Dr. Thomas Westermann for his acceptance to involve in my project, and become my second supervisor .

I would also like to extend my thanks to the professors in Karlsruhe University of Applied Sciences for teaching me that I was able to deliver several sections of this project.

Thank you for giving me a chance to study in Karlsruhe and wish you all the best!

1. Overview:

1.1. Abstract:

Time has been changed, the expansion of information makes a big change in industry or health care, service or transportation. As a result, the concept of IoT was introduced and becomes more popular, it appears and is often an integral part of all aspects of life and technology. For example, many years ago, to measure heart rate parameters, patients had to make an appointment and visit a doctor, which is troublesome when compared to today, when IoT was applied, patients only need to wear an end-device, which has the necessary sensors and connection (WiFi, 4G). Then, the doctor can update his heath continuously, real-time without the two even having to see each other, that's really is a revolution.

To do that, the connections between devices have to be established for data sharing. In the previous example, it is the connection and communication between patient's end-device and doctor's laptop/smartphone. This is also correct with a bigger system, which includes many smaller systems, when each small system – likes a device in the big system, need to "talk" to each other to delegate tasks or receive the input for its tasks.

Hence, data contribution is the most important. Additionally, the way to contribute information, must be reliable, stable, and lightweight as well. As a result, MQTT protocol came to solve all that problems, which was invented in 1999 by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom, now Cirrus Link). Nowadays, it is the most commonly used messaging protocol for the Internet of Things (IoT).

1.2. Literature review:

1.2.1. Conceptions:

MQTT Protocol

> *"MQTT is a Client Server publish/subscribe messaging transport protocol. It is light weight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium."*

Citation from the official MQTT 3.1.1 specification. [1]

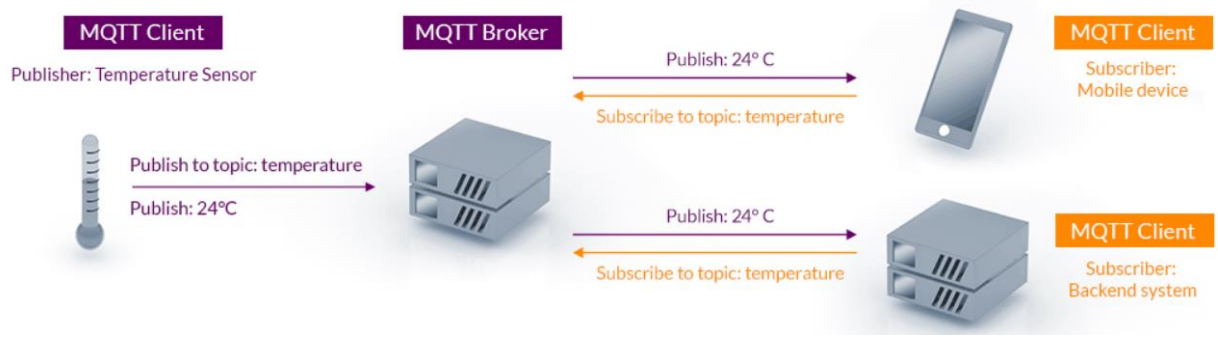


Figure 1. MQTT Architecture (Pub/Sub Model) [2]

MQTT protocol follows publish/subscribe model, which is also a client-server architecture base on TCP/IP. In MQTT, it defines MQTT Client (client) and MQTT Broker (server). When a Client want to send its information to another Client, the data is not transmitted directly between two devices. Instead, the Client will connect and send the information to Broker – that action is called Publishing and the Client is called Publisher. After that, Broker will pass that data to the Client - which asks for that information – the action is called Subscribing and the Client is called Subscriber.

That means in MQTT protocol, every Clients do not need to know each other, they just need to deal with Broker and Broker will try to manage any communications between Clients.

The essential properties of MQTT that meet the needs of IoT applications are conceptual, but for a deeper understanding, we will compare it to a highly popular protocol, HTTP. Both are based on TCP/IP, which has lately become the most widely used data transmission technology. They also share a Client-Server architecture, as seen in the table below: [3]

	MQTT	HTTP
Architecture	Publish subscribe	Request response
Command targets	Topics	URLs
Underlying Protocol	TCP/IP	TCP/IP
Secure connections	TLS + username/password (SASL support possible)	TLS + username/password (SASL support possible)
Client observability	Known connection status (holding connection)	Unknown connection status
Messaging Mode	Asynchronous, event-based	Synchronous (any request needs to wait for an response)
Message queuing	The broker can queue messages for disconnected subscribers	Application needs to implement
Message overhead	2 bytes minimum. Header data can be binary	8 bytes minimum (header data is text - compression possible)
Message Size	256MB maximum	No limit but 256MB is beyond normal use cases anyway.
Content type	Any (binary is normal)	Text (Base64 encoding for binary)
Message distribution	One to many	One to one
Reliability	Three qualities of service (QoS)	Has to be implemented in the

		application

Table 1. Comparation about features between MQTT and HTTP Protocol [4]

Besides, by trying to connect to Google IOT Core, we can measure performance of 02 protocols:

	MQTT Bytes	HTTP Bytes
Establish connection	5572	2261
Disconnect	376 (Optional)	0
For each message published	388	3285
Sum for 1 message	6336	5546
Sum for 10 messages	9829	55460
Sum for 100 messages	44748	554600

Table 2. TCP message overhead [4]

The MQTT Protocol consumes significantly less bytes then HTTP, that means the application can save a lot of bandwidth and energy.

Finally, we measure the response time:

No. messages in a connection cycle for MQTT	MQTT avg. response time per message (ms) (QoS 1)	HTTP avg. response time per message (ms)
1	113	289
100	47	289
1000	43	289

Table 3. Response time per message [4]

According the measurement above, we can confirm that the features of MQTT Protocol, are mostly more suitable with IoT applications than HTTP.

MQTT Broker:

The author will implement the most important, the heart of a MQTT network - MQTT Broker - in this project. In a nutshell, MQTT Broker is the location where all network packets must be stopped. Furthermore, it must handle many messages and tasks concurrently: receiving all messages from clients, checking the packet, routing the message to the destinations... In other words, MQTT Broker is a hub where all of the main tasks of the MQTT network are carried out. To summarize, whether or not a MQTT network can function properly is largely determined by its Broker.

Concisely, MQTT Broker need to handle these basic tasks:

- ✓ Receiving all messages,
- ✓ Filtering the messages,
- ✓ Determining who is subscribed to each message,
- ✓ Sending the message to these subscribed clients

Otherwise, there are some other tasks of a MQTT Broker:

- ✓ Holds the session data of all clients
- ✓ Authentication and authorization of clients

Therefore, it is important that your broker is highly scalable, integratable into backend systems, easy to monitor, and (of course) failure-resistant.

1.2.2. Topic review:

There are many documents, including articles, open sources, blogs, etc., that explain the MQTT Protocol, therefore it can be difficult for researchers to get started. Fortunately, a website [5] has been created to compile and summarize the vast majority of the information you need to know about the MQTT Protocol, including its specifications, standards, applications, and usage. However, it frequently focuses on practical rather than intellectual features. It is appropriate for this thesis and will be referred.

The website [5] is a fairly complete and reliable aggregator of MQTT concepts and applications, softwares, and libraries in progress (both open sources and commercial products). Obviously, individual projects with limited resources and applications are not included, which are not widely publicized, so this will be an important reference and will be cited throughout this thesis.

Generally, MQTT Brokers can be classified into two types: Managed Brokers và Self-Hosted Brokers.

Managed Brokers do not require any server configuration to enable MQTT communication. You can use Managed Broker services to use their hosted brokers for your system. A good example of a managed MQTT Broker is AWS IoT Core [6].

To compare, **Self-hosted MQTT Brokers** necessitate the installation of the broker on your own VPS or server with a static IP address. The installation process is simple, but managing, securing, and scaling the brokers necessitates extensive knowledge of the system. There are several open-source MQTT brokers available, including mosquitto [7] and hivemq [3].

A Managed Broker service makes sense if you need to quickly construct a prototype or proof-of-concept (POC) and do not want to spend time managing the infrastructure and protecting the connections. You can access a Broker that is ready to use in just a few clicks. Though, managed brokers can have some disadvantages. For instance, you will be charged based on the quantity of data packets sent (the pricing model varies from service provider to service provider). The majority of the Broker's settings are out of your hands; you may only make changes that the vendor permits. Data transfer and packets/second or packets/minute are frequently capped by managed broker providers, which might impede communication.

A Self-hosted Broker can be the best option if you try a Managed Broker but the cost or control restrictions caused an issue, that also means you are required programming and networking skills. On contrast, with a Self-hosted Broker, you can quickly implement your rules, scale the system as you see fit, and configure it anyway you choose.

Here is an overview of some of the most popular options.

Type	Type	WebSocket Support	SSL Support	Scalability
AWS IoT Core MQTT	Managed	Dynamically assigned	Yes, port=443	Yes, port=8883
Mosquitto	Self-hosted and Managed	test.mosquitto.org	Yes, port=8081,8080	Yes, port=8883,8884
Mosca/Aedes	Self-hosted and Managed	test.mosca.io	Yes, port=3000	Yes, port=8883
HiveMQ	Self-hosted and Managed	broker.hivemq.com	Yes, port=8000,443	Yes, port=8883
VerneMQ	Self-hosted and Managed	self-assigned	Yes, port=9001,9002	Yes, port=8883
Azure IoT Hub	Managed	Dynamically assigned	Yes, port=443	Yes, port=8883
EMQ X	Self-hosted	Self-assigned	Yes, port=8083, 8084	Yes, port=8883
ejabberd	Self-hosted and managed	Self-assigned and dynamically allocated	Yes	Yes

Table 4. List of Popular MQTT Brokers

Generally speaking, the analysis above highlights both the pros and cons of the published MQTT Brokers projects. There is a comment stating that the aforementioned programs are either paid or opened source - but challenging to understand and modify to the demands. All the brokers have been found as good performance, but they are not easy to customize in mostly cases.

In order to create a MQTT Dashboard that can further configure for other applications, the author will create a new program using a different paradigm based on the Actor Model (to be discussed later). Self-hosted Broker would be the preferred approach because it is simpler to understand and control, lighter, more convenient, and does not impose fees.

2. Introduction of thesis:

2.1. The approach of this thesis:

Based on the idea of developing such a MQTT Dashboard, the author will implement a MQTT Broker written in Scala - a powerful language supported by the AKKA toolkit, with a set of tools powerful processing engine, and very good at scalability and self-healing. As stated in the introduction to MQTT Broker, this will be useful in the role of a broker, who must always connect to many devices, handle many processes, and must always be stable, but easily extendable.

The development of user functions, on the other hand, will be based on the Play Framework - a framework built on top of the AKKA model that allows it to inherit the power of AKKA and is also the framework used for the Scala language.

Thanks to the developer Butaji, who has released an open source (MIT license) from the webpage. [8]

This project was developed 7 years ago and is almost impossible to run at the present time, when the libraries are no longer supported and downloaded from the internet. However, it also provides logical directions for this project, especially the core MQTT Broker.

2.2. Thesis aims and objectives:

Aims:

- ✓ To develop a MQTT Broker Core using Scala and AKKA toolkit.
- ✓ To develop a Dashboard (Web Client for User Interface) using Scala and Play Framework.
- ✓ To acquire and implement AKKA model for MQTT Protocol.
- ✓ To achieve the reliability of the program in real IoT applications.

Objectives:

- ✓ Developing a MQTT Broker Program.
- ✓ Developing a Website with User Functions and User Interface interacts with MQTT Broker Core to monitor the devices in network.
- ✓ Programing language: Scala.
- ✓ Model/Framework: AKKA & Play.

Diagram:

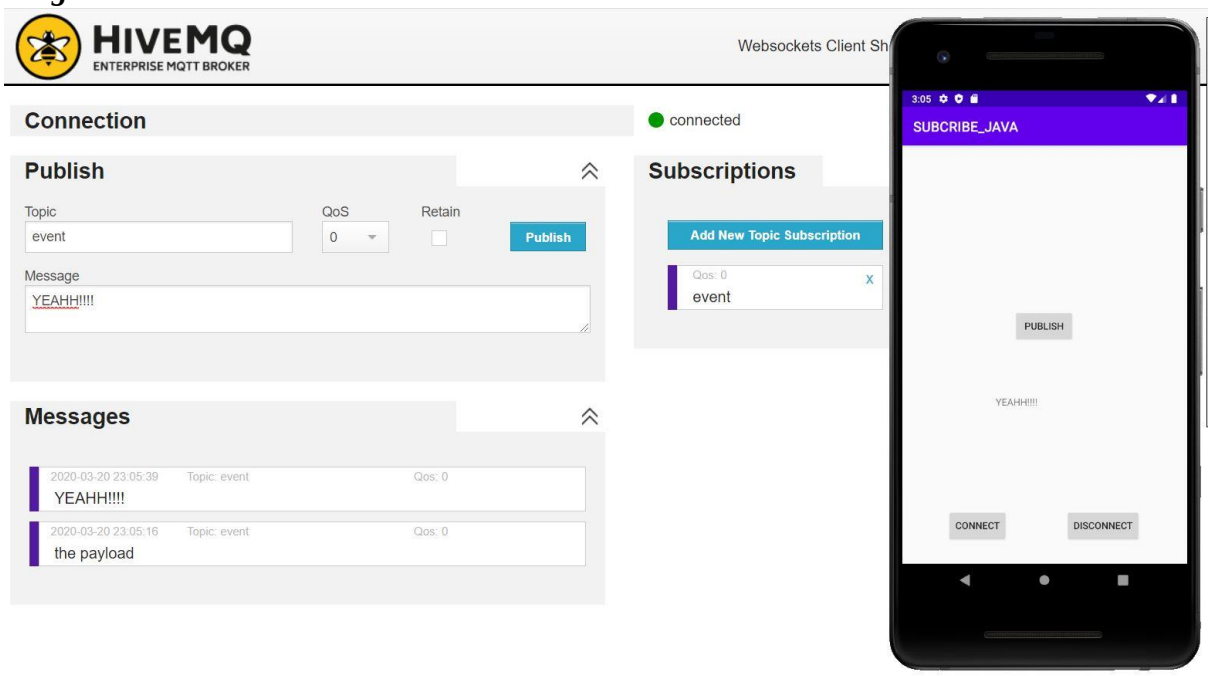


Figure 2. Illustrating images of dashboard

To compare with previous projects, this is a new approach, base on Scala language and AKKA model could be a good perspective for other projects using Scala or AKKA.

3. Tools and Methods:

The combo Scala, Akka toolkit and Play Framework will come together to implement the ideas in this project.

3.1. Introduction of Scala:

Scala is a powerful statically typed general-purpose programming language that can be used for both object-oriented and functional programming. Designed to be concise,[9] many of Scala's design decisions are intended to address Java's criticisms.[7]

Scala source code can be compiled into Java bytecode and then executed on a Java virtual machine (JVM). Scala and Java share language interoperability, allowing libraries written in either language to be referenced directly in Scala or Java code.[10] Scala, like Java, is object-oriented and employs a syntax known as curly-brace, which is similar to the language C. Since Scala 3, there is also the option of using the off-side rule (indenting) to structure blocks, which is recommended. According to Martin Odersky, this was the most fruitful change introduced in Scala 3.[11]

Advantages:

- ✓ Easy to Pick Up
- ✓ Pretty Good IDE Support
- ✓ Scalability
- ✓ Highly Functional
- ✓ Code cleaner

3.2. Introduction of AKKA:

AKKA is a toolkit that includes a collection of modules and libraries for building distributed applications with the Actor Model as the programming model. AKKA supports the Scala and Java programming languages and has an API for each of them. This toolkit is used by companies such as Verizon, Intel, PayPal, Norwegian Cruise Lines, and Samsung.

Advantages:

- ✓ Event-driven: With Actors, requests can be done asynchronously and non-blocking operations exclusively
- ✓ Scalable: By message passing and location transparency, adding nodes without having to modify the code is possible.
- ✓ Resilient: Akka fault tolerance is to encounter errors a self-healing system.
- ✓ Responsive: Akka’s non-blocking, message-based strategy helps to give quick feedback to request.

AKKA documentation is thorough and provides a comprehensive understanding of the many ideas and tools available. The majority of toolkit implementations are based on field research and industry expertise. Lightbend - the firm driving AKKA's development, is a commercial company that provides corporate software solutions for distributed systems and the cloud environment. This business is also responsible for the creation of other frameworks and platforms, like the Play framework and the Lagom framework5, which employ AKKA as an underlying technology.

3.2.1. AKKA Actor:

Actor programming model

The Actor Model consists of a set of actors, which are isolated, concurrent, and solely interacted through a network with a transparent message-passing technique. [9] The Actor Model was introduced by Carl Hewitt in 1973. [10] The model was designed to provide a general paradigm for concurrent computing in a highly concurrent and parallelizable distributed environment. At a higher level, the model is straightforward and allows for a high degree of parallelism. An Actor is the primary unit of the computing model. An Actor is a type of entity that may connect with other actors via network communications. [11] An actor can also generate additional actors in the network based on the needs. In this situation, the creator actor will be the "parent actor," and the created actors will be the "child actors." The following are the key benefits of the Actor Model.

- ✓ An Actor Model extends the benefits of object-oriented programming by decoupling business logic and control flow.
- ✓ An Actor Model enables the decomposition of a system into independent, autonomous, and interacting components that can function in parallel.

The race situation is avoided because one actor cannot corrupt the state of the other actors in the network. Each actor uses the spawn procedure to add a new actor to the network. This method is frequently used to spread workloads using a divide and conquer strategy. An actor separates incoming tasks into a set of sub-tasks and distributes them simultaneously among freshly formed actors using the spawn operation to speed up processing and improve response time of IoT applications. During the processing or analysis of an IoT application, each actor watches the other actors in the network to discover and propagate faults in the distributed environment. Bi-directional connections can express stronger coupling between network actors for improved message conveyance and prevent incorrect states during run-time. An actor embodies the three qualities listed below.

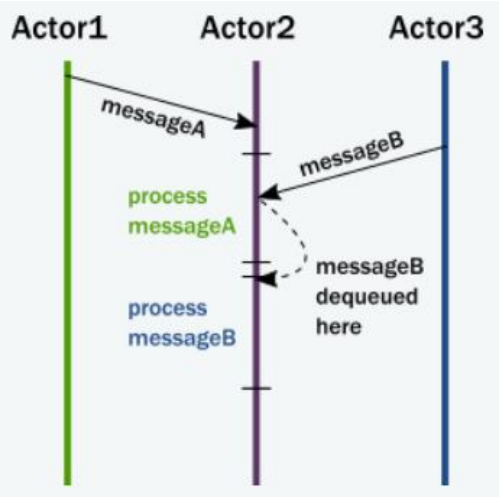


Figure 3. The message passing and processing flow of multiple actors [12]

- ✓ **Information processing**: Partial/complete processing of applications for their behavior.
- ✓ **Storage**: Save an actor's current state.

- ✓ **Data Communication**: Using the message passing approach, send the message to the remaining network actors.

Scala is a programming language that includes the actors for the standard distribution with the AKKA framework.

Actor model of computation

In the OOP Paradigm, an actor, like an object, has state and behavior. However, there are several constraints in the Actor Model that provide assurances while performing out calculations. [11] The state is totally owned by the actor and cannot be shared or accessed by other players in the system. This means that locks or other sorts of synchronization methods are unnecessary in a multi-threaded system. An actor can alter its state in response to a message or execute some calculation based on the message. The computation capabilities of the actor define its behavior.

One of the core ideas of the Actor Model is message passing. Fig. 3 depicts the message passing processing flow of various actors. An actor can respond to perceived signals sequentially and analyze each message separately, i.e. one at a time. While each actor processes the incoming messages sequentially, the other actors can operate concurrently with each other, allowing the network to handle numerous messages at the same time. It is the sole permitted method of communication between network participants. The following are the stages an actor takes when processing a message.

- ✓ The actor adds the current incoming message to the queue's end.
- ✓ If the actor is unavailable and the message was not scheduled for processing, the actor marks the message as ready to process.
- ✓ A secret scheduler pulls the most recent ready message from the queue and begins processing it.
- ✓ The actor updates current state information and sends it to another actor.
- ✓ Finally, the actor takes the message out of the queue.

In order to carry out the above operation, an actor must have the following properties:

- ✓ **A mailbox** queues incoming messages on a first-come-first-served basis.
- ✓ **A behavior** includes internal variables, an actor's state, and so on.
- ✓ **Incoming messages** contain data for expressing single or more methods and their parameters.
- ✓ **A processing environment** takes the actors who have certain messages to react to and calls their message handling codes.
- ✓ **An address** indicates an actor who will allocate incoming messages.

Because each actor may only process one message at a time, the invariant of an actor can be maintained without synchronization. This occurs automatically without the use of locks. One of the model's primary successes is the decoupling of the actor from the message-sending mechanism, which may be done asynchronously. An actor in a network can only communicate with other linked actors. Direct physical attachment, memory/disk access, network address, or E-mail address can all be used to connect the actors.

The addresses of the actors will vary depending on the sort of connections. In the event of a physical link, it may be a MAC address or just a memory address. Messages are provided

using all reasonable attempts. It is the receiver's job to handle a message once it has been sent by an actor. This is the crucial component that allows a message to be decoupled from the sender actor. This method of communication is also known as "fire and forget."

The Actor Model is therefore an abstract construct based on some axioms that describe the model's behavior and structure. Several qualities and mechanisms are at work behind the scenes. Implementations of the model should follow those principles and may utilize other ideas on top of them to reveal the model's behavior in a practical fashion.

3.2.2. AKKA Stream:

Reactive Stream

In an asynchronous system, dealing with data streams—particularly "live" data whose volume cannot be predicted—requires special care. The most obvious difficulty is that resource consumption must be managed so that a rapid data source does not overwhelm the stream destination. Asynchrony is required to allow for the concurrent usage of computer resources, whether on collaborating network hosts or numerous CPU cores inside a single system. [12]

The primary purpose of Reactive Streams is to manage the exchange of stream data over an asynchronous boundary (imagine transferring components on to another thread or thread-pool) while preventing the receiving side from being compelled to buffer arbitrary quantities of data. In other words, back pressure is an essential component of this paradigm in order to keep the queues that mediate between threads constrained. Because the benefits of asynchronous processing would be negated if back pressure communication was synchronous, care must be taken to require entirely non-blocking and asynchronous behavior of all components of a Reactive Streams implementation. [12]

The goal of this specification is to enable the design of multiple conforming implementations that, by adhering to the rules, will be able to interoperate easily, keeping the aforementioned advantages and features over the whole processing graph of a stream application. [12]

AKKA Stream

Actors in AKKA may also be thought of as dealing with streams: they send and receive a sequence of messages in order to convey knowledge (or data) from one location to another. In that process, it time-consuming and error-prone to implement all of the necessary measures in order to achieve stable streaming between actors, because in addition to sending and receiving, we must also avoid overflowing any buffers or mailboxes in the process. Another drawback is that Actor messages might be lost and must be resent in such cases. If this is not done, holes will form on the receiving side.

That is the reason for AKKA Stream API. The goal is to provide an easy and secure approach to create stream processing configurations that can then be executed fast and with little resource usage—no more OutOfMemoryErrors. In order to achieve it, these streams must be able to control the amount of buffering they utilize, as well as slow down producers if consumers cannot keep up. This feature is known as back-pressure, and it is at the heart of the Reactive Streams effort, of which AKKA is a founding member. This implies that the

difficult challenge of propagating and responding to back-pressure has already been included into the architecture of Akka Streams, giving users one less thing to worry about. [13]

The Akka Streams API is entirely independent of the Reactive Streams interfaces. While Akka Streams is concerned with the formulation of transformations on data streams, the goal of Reactive Streams is to create a standard method for moving data across an asynchronous boundary without loss, buffering, or resource depletion.

3.3. Introduction of Play framework:

Play is a very simple and straightforward Web Framework. It was designed to help you make changes more quickly and easily, with less stress on you.

Thanks of its smooth and user-friendly interface, not to mention several options for maximizing your computer's resources - CPU, RAM - it's simple to grow the software you're building. It is intended for developers that want to create contemporary web and mobile apps.

Play is developed on top of the AKKA toolkit, a popular open source toolkit that runs on top of the JVM. It has the same fundamental tools and functionality as the AKKA toolkit, but operates in a more user-friendly manner that allows you to simply create, design, and test the apps you're building. Many developers have remained loyal to it, citing how it has increased their productivity due to its simplicity and ease of use.

Advantages:

- ✓ Huge productivity improvement
- ✓ Workflow is easy
- ✓ Flexible Tool
- ✓ Everything works from the moment you start
- ✓ Effective resource management unit
- ✓ Easy to scale software

4. Developing of MQTT Broker:

4.1. Software design of programs:

The previous section discussed AKKA Toolkit and why AKKA was chosen to develop MQTT Broker for this project. In this section, we will clarify the ideas for implementing the MQTT Broker program.

Actors Model of Program:

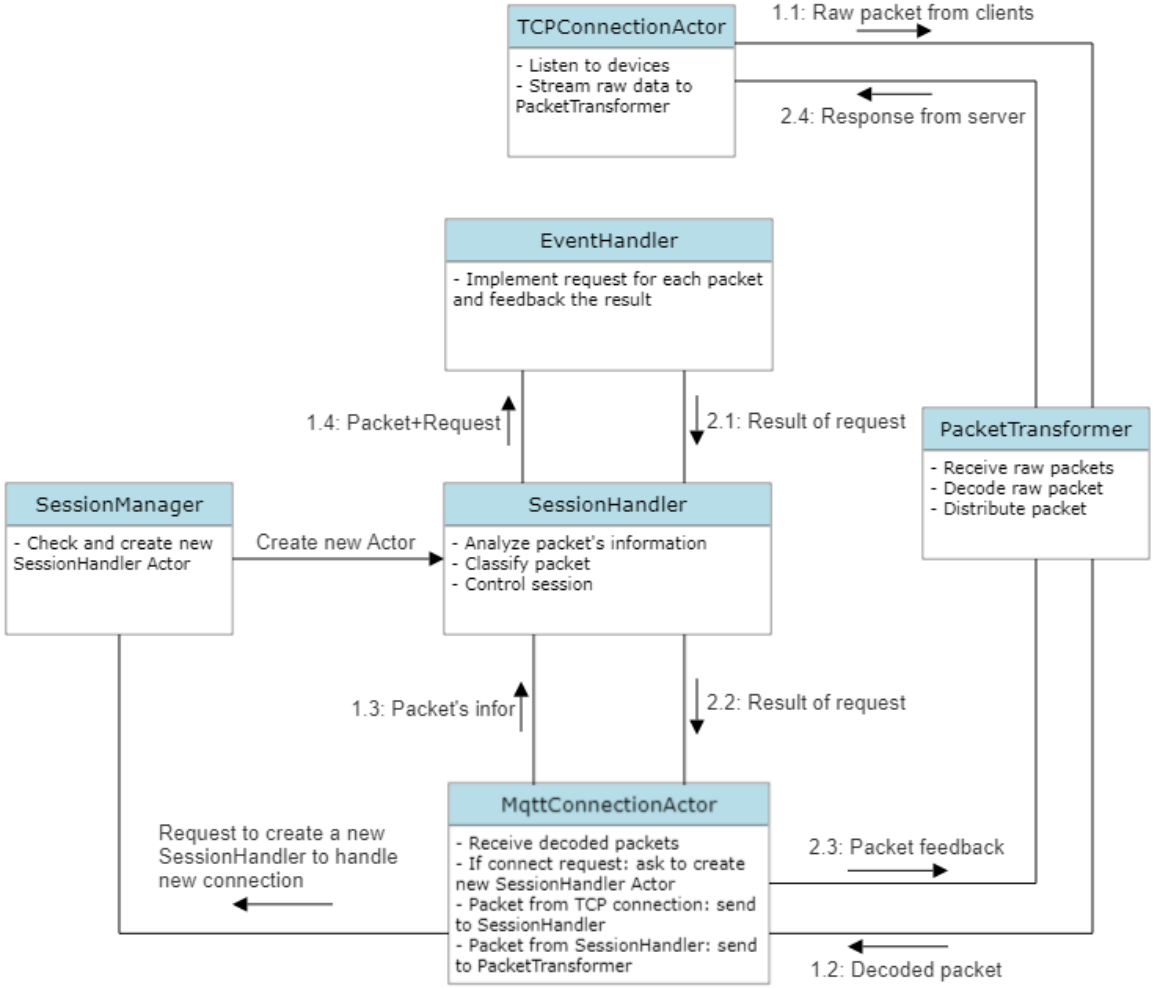


Figure 4. Actor Model for Core MQTT Broker (UML Communication diagram)

The figure above is the project's official model, displaying the functions as well as communication between actors via the Actor model - Hierarchy.

There is a root actor, which serves as the root for all actors in the system. Because of the centralized management but independent handling of connections – which representatives for a Client or Device, groups of Actors will be born to handle each connection. Actors will be managed using the Hierarchical model, which aids in the accurate management of executions and errors, resulting in high system stability. For more detail, these are explanations:

- **TCPConnectionHandler**: It handles all connection requests and communicates with peripheral devices by opening a socket and listening; for each valid connection, it spawns ***PacketTransformer*** actor for that connection. AKKA Stream is created and used to transmit data in streams and queues between them rather than sending messages directly; this allows for more accurate and reliable packet delivery and management.
- **PacketTransformer**: Every time a connection is established, this actor is born to be in charge of delivering packets from the connection to other actors so that they can handle tasks or return ACK packets to clients. Its major function is to decode and transport packets in the system in the manner of a distributed station.

- **MqttConnectionActor**: This actor will be spawned at the same time as the actor ***PacketTransformer***; it will receive packets from the ***PacketTransformer*** and analyze them. In case the packet is a valid Connection Request, it will request (ask) to create an actor handles this connection; otherwise, it will forward the packet to actor ***SessionHandler*** or PacketTransformer for further processing. Cases that are invalid will be rejected.
- **SessionManager**: The ***MqttConnectionActor*** will then check to see if an actor SessionHandler is already handling this connection; if not, the ***SessionManager*** will create a new Actor ***SessionHandler*** to handle this connection.
- **SessionHandler**: This actor is in charge of analyzing and classifying packets sent over the connection, including checking the Header and Control Flags ,etc. After classifying the packet and gathering the required information, the actor will send (tell) the data to actor ***EventHandler***, which will handle the tasks required by each packet, to get feedback. Besides, it also generate ACK packets (or PUBCOMP/PUBREC) to send back to connection (or client) via ***MqttConnectionActor***.
- **EventHandler**: once the packets have been classified, they will be sent (tell) to this actor who will execute the packet's request and return the result to the ***SessionHandler***.

The table below summarizes the actor descriptions and their interactions with other actors.

No.	Actor	Description	Related Actor
1	TCPConnectionHandler	This actor handles TCP connection and listen to device. Receive and transmit raw packet from devices.	→ SessionManager → EventHandler ↔ PacketTranformer
2	PacketTransfomer	This actor handles packets which are sent via TCP and filter/decode/transfer them between other actors (inbound) or client (outbound)	↔TCPConnectionHandler ↔MqttConnectionActor
3	SessionManager	Check and create new SessionHandler to control a new connection	← MqttConnectionActor
4	MqttConnectionActor	This actor handles packets inbound: + Receive decoded packet → analyze + Decide action for each packet → PacketTransformer + Separate & send packet Infor → SessionHandler to process + Receive feedback from SessionHandler → send to PacketTransfromer	↔PacketTransformer ↔ SessionHandler
5	SessionHandler	Control operation process of each connection: + Get packet's infor → classify packet + After classifying → Tell EventHandler to implement the task + Get the data after Event handled + Send feedback MqttConnectionActor	↔ EventHandler ↔ MqttConnectionActor
6	EventHandler	Implement the packet's requirement (task)	↔ SessionHandler

Table 5. Description of Actors in MQTT Broker Program

Project's structure

The architecture of the project is showed below

```
src                                  → Application sources
 └main                               → Main sources
   └Core                             → Package contains core functions of MQTT
     └Connection                     → Package contains actors handle the communications
       └ServerActor.scala            → Actor handles TCP connections
       └MqttConnectionActor.scala    → Actor handles handles packets inbound
       └PacketTransformer.scala      → Actor transmits packets between actors and clients
     └PacketHandler                  → Package contains actors process packets
       └Packets.scala                → Actor processes the packets
       └PacketHelper.scala           → File contains supported function to process the packets
   └Session                          → Package contains actors manage connections
       └EventHandler.scala           → Actor implements the packet's requirement
       └SessionHandler.scala         → Actor controls operation process of each connection
       └SessionManager.scala         → Actor checks and creates new SessionHandler
test                                 → Source folder for unit or functional tests
build.sbt                            → Application build script
conf                                 → Configurations files and other non-compiled resources
 └application.conf                   → Main configuration file(optional)
project                              → sbt configuration files
 └project                            → Marker for sbt project
 └build.properties                   → Marker for sbt project
 └target                             → Generated stuff
target                               → Generated API docs
```

Dependencies

Akka actor, *akka stream* and *akka-http* are used in the program as mentioned above. *Scodec library* is also used to handle the binary-computing. To do that, the sbt file is configured:

```
val AkkaVersion = "2.6.19"
val AkkaHttpVersion = "10.2.9"
val CodecVersion = "1.11.9"

libraryDependencies ++= Seq(
  //for core
  "com.typesafe.akka" %% "akka-actor" % AkkaVersion,
  "com.typesafe.akka" %% "akka-stream" % AkkaVersion,
  "com.typesafe.akka" %% "akka-http" % AkkaHttpVersion,
  "org.scodec" %% "scodec-core" % CodecVersion,
```

For testing, akka test library is added.

```
//for testing
"org.scalatest" %% "scalatest" % TestVersion % Test,
"com.typesafe.akka" %% "akka-actor-testkit-typed" % AkkaVersion % Test,
"org.scalatest" %% "scalatest-wordspec" % TestVersion % "test",
"com.typesafe.akka" %% "akka-slf4j" % "2.6.5",
"ch.qos.logback" % "logback-classic" % "1.2.11"
```

To view full source code, please access the link [github repository of project] [14].

4.2. Implementing the logic:

In the preceding section, we used the Actor Model to model the program for MQTT Broker. Following that, we will apply this model to a specific program.

However, quoting and explaining thousands of lines of code is time-consuming and unnecessary. As a result, to ensure the report's conciseness, the author will only point out and analyze relatively complex programming techniques. In addition, the program's explanations is commented in the source code comments.

Finite State Machine (FSM):
FSM is the programming method used throughout (Finite State Machine). This is a tried-and-true method with high reliability.

A FSM can be described as a set of relations of the form [15]

$$State(S) \times Event(E) \rightarrow Actions\ (A),\ State(S')$$

These relations are interpreted as meaning [15]

> *If we are in state S and the event E occurs, we should perform the actions A and make a transition to the state S'.*

Thanks to AKKA Toolkit, implementing FSM is more concise and reliable.

The FSM is applied to control the process in each actor. By forecasting and planning states, programming becomes more accurate and trustworthy.

The table below shows the FSM for **MQTTConnectionActor**:

No.	State	Event	Action	Next State
1	Wait (Idle)	Receive: Connect Packet	Ask for SessionManager	Active (Connected)
2	Wait (Idle)	Receive: Unvalid packet	Send Closing (connection)	Wait (Idle)
3	Active (Connected)	Receive: Packet from Session	Send back to Connection	Active (Connected)
4	Active (Connected)	Receive: Packet from Connection	Send to Session	Active (Connected)
5	Active (Connected)	Timeout	Send Closing (connection)	Active (Connected)
6	Active (Connected)	Bad packet	Send Closing (connection)	Active (Connected)

Table 6. FMS for MQTT Connection Actor

State machine diagram:

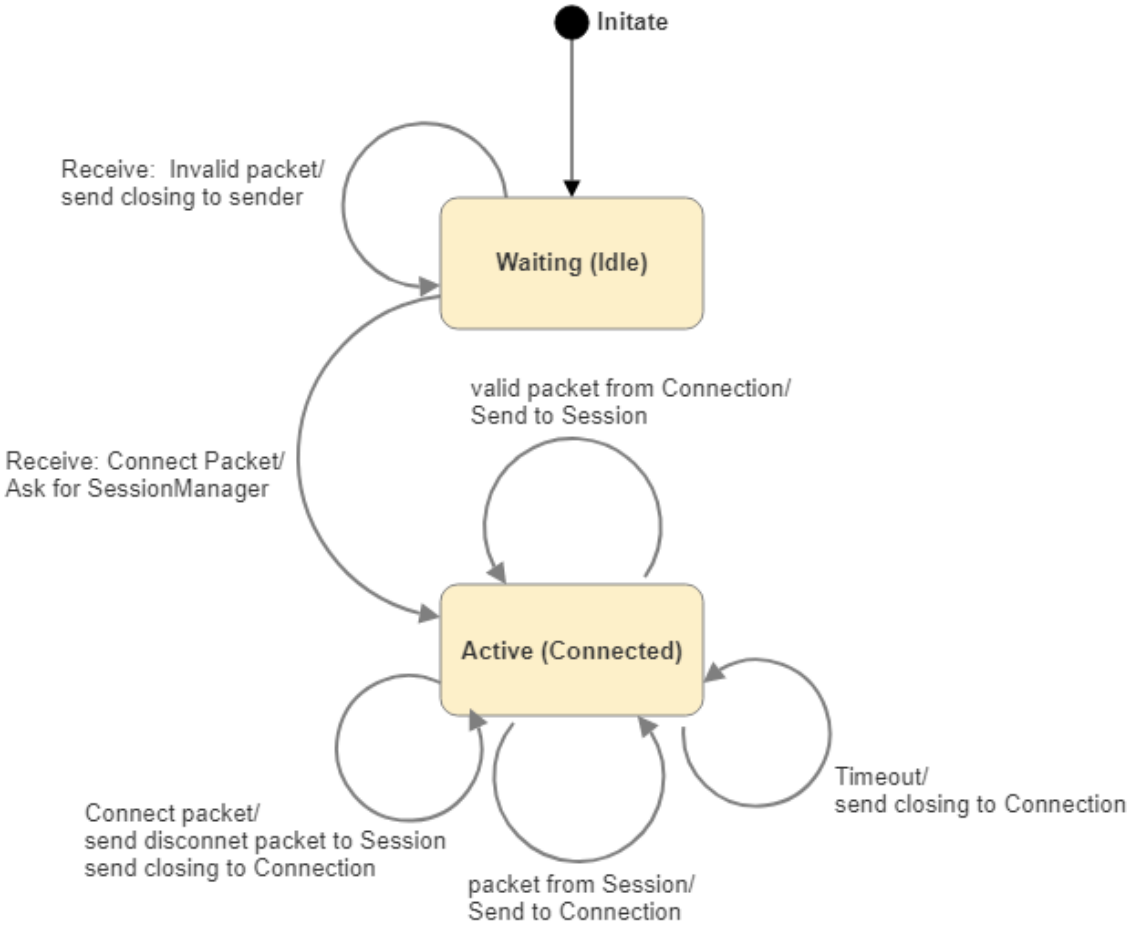


Figure 5. MqttConnection Actor FSM diagram

The FSM for **SessionHandler** is revealed:

No.	State	Event	Action	Next State
1	WaitingForNewSession	Receive: Connect Packet	Take the infor Send CONNACK to PacketTransformer	SessionConnected
2	WaitingForNewSession	Receive: bad packet	Send error → Closing	WaitingForNewSession
3	WaitingForNewSession	Receive: PublishPayload	Check qos > 0 → EventHandler	SessionConnected
4	SessionConnected	Timeout	Close Connection	SessionConnected
5	SessionConnected	Receive: Sub	Packet → EventHandler SubAck → MqttConnectionActor	SessionConnected
6	SessionConnected	Receive: Publish	qos = 1 → EventHandler + PubAck qos = 2 → Pubrec + EventHandler	SessionConnected
7	SessionConnected	Receive: PubAck	stay	SessionConnected
8	SessionConnected	Receive: PubRec	→ send PubRel	SessionConnected

9	SessionConnected	Receive: PubRel	→ send PubComp	SessionConnected
10	SessionConnected	Receive: PubComp	stay	SessionConnected
11	SessionConnected	Receive: Sub	Packet → EventHandler	SessionConnected
12	SessionConnected	Receive: Unsub	Packet → EventHandler	SessionConnected
13	SessionConnected	Receive: Ping	Stay	SessionConnected

Table 7. FMS for Session Handler Actor

State Machine diagram:

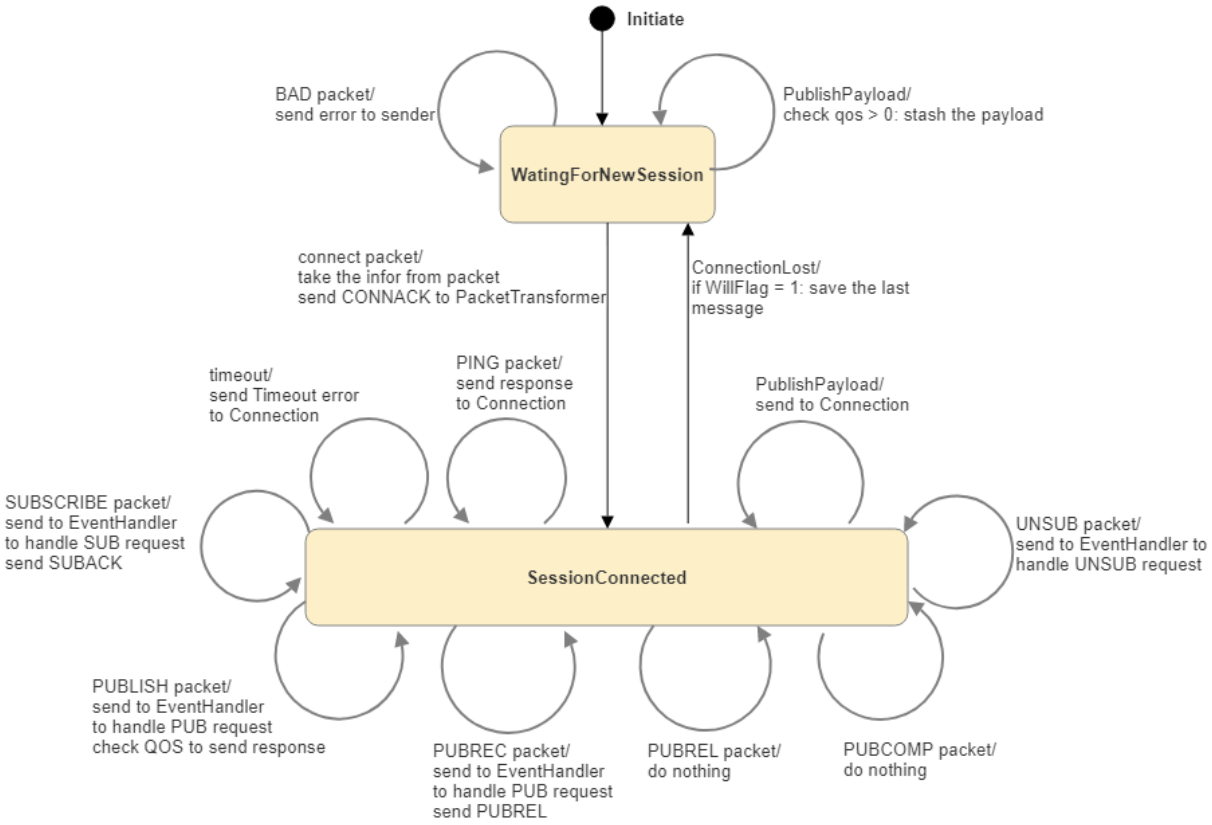


Figure 6. SessionHandler Actor FSM diagram

4.3. Testing and Result:

The intriguing aspect of AKKA Toolkit is that it provides both testing functionality for our applications, allowing us to test functions independently while also testing a group of functions or even the entire system with various scenarios.

The AKKA test toolkit is used to check every single test case in order to guarantee that every single function works flawlessly. As a result, each test case passing indicates that the function works properly.

First of all, the test cases should be predicted and defined. Then, the code blocks to run those scenarios have to be written to illustrate the situations. For example:

```
class BusSpec extends TestKit(ActorSystem("BusSpec")) with ImplicitSender
  with AnyWordSpecLike
  with Matchers
  with BeforeAndAfterAll  {

  "EventBus actor can do:" should {

    "simple pubsub - ok!" in {

      val bus = system.actorOf(Props[EventBusActor])

      bus ! BusSubscribe("greetings", self)
      bus ! BusPublish("time", "123")
      bus ! BusPublish("greetings", "hello")

      expectMsg(PublishPayload("hello", false))

      expectNoMessage(UserTimer.wait_time)
    }
```

In here, the test simulates a simple task for EventHandler actor. As mentioned, this actor need to classify the topic and do the task as requests. In this case, the requests are to subscribe a topic name "greeting" then publish two topics name "time" and "greeting" with the value "123" and "hello" respectively. If the task has been done correctly, the `expectMsg()` will receive the payload "hello" and the comment "simple pubsub – ok!" will be marked by green tick.

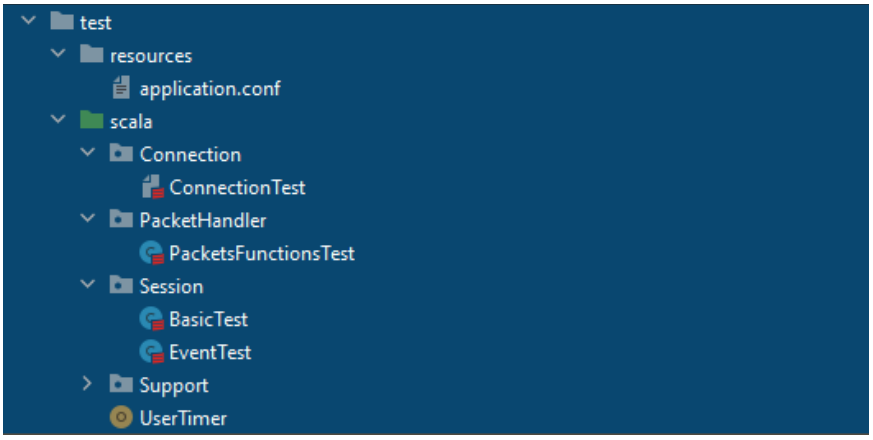
The test also can combine burst of code blocks, which represent for many test cases. If all the test passed, that means the actor works well.

4.3.1. Unit Tests:

To begin testing, the unit test is used to confirm that all functions can be executed appropriately. In general, this approach may be used to examine the actor(s) by constructing scenarios and verifying the actor(s)'s reactions.

For each test, the condition(s) and the expected message(s) are defined. In the case we get the correct information, the test is passed and that means that function is working well.

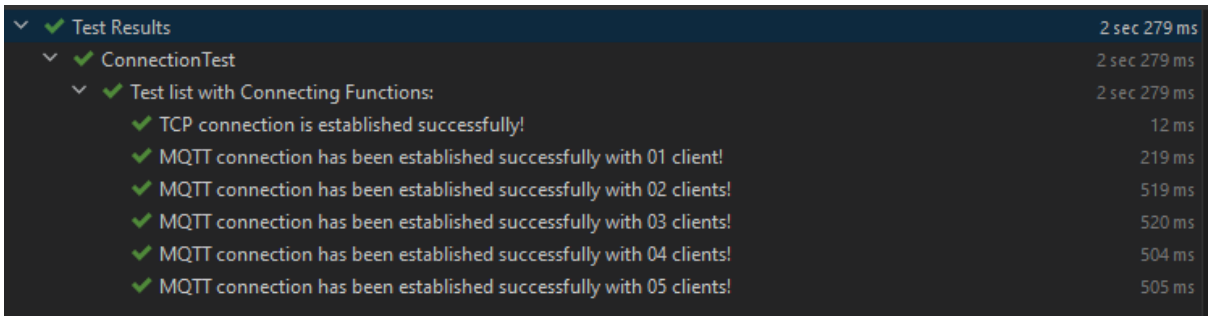
The testing is splitted into 04 test for 03 packages, including:



TCP Connecting Test

This is the very first step to check the execution of server. When the server is started, clients are created to connect to server, as well as send simple packets to the it and attempt to get response packets.

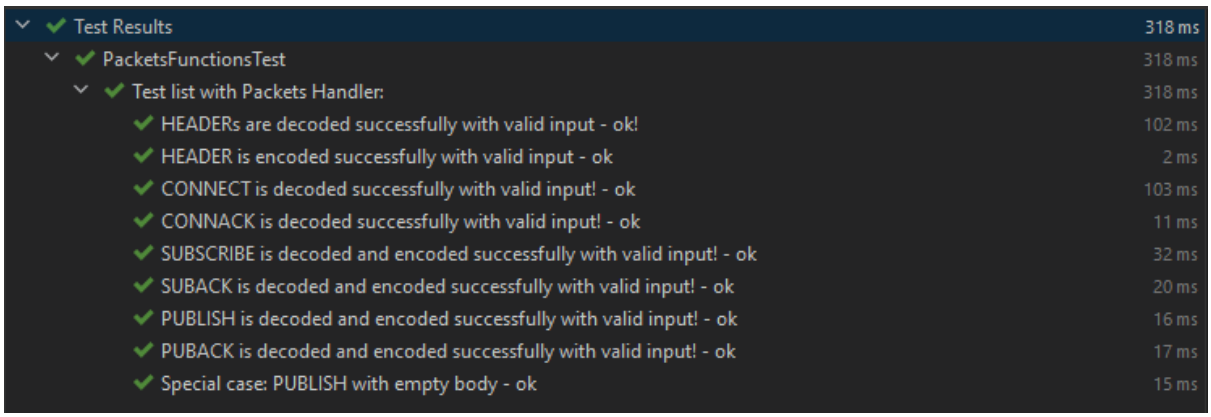
The test results as below:



Conclusion: TCP and MQTT connections (for 01 client and more) are established successfully, all the tests pass!

Packet Handling Test

This test will assist in determining whether the packets are successfully encoded/decoded. In this, some random packets are decoded and encoded, then check with the defined format.

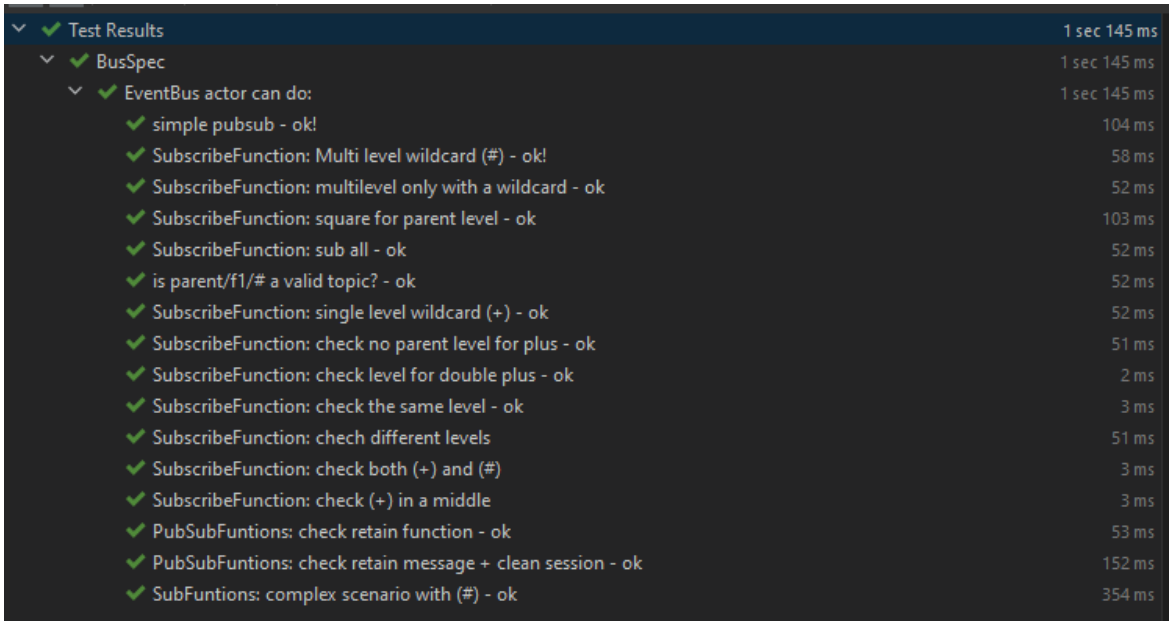


Conclusion: Encoding and decoding are success in all test cases.

Event Handling Test

The deconstruction of the packet's requirements and the execution of those requirements will be done in this test to determine if the requirements are satisfied or not.

The test results are showed below for **EventHandler** actor:

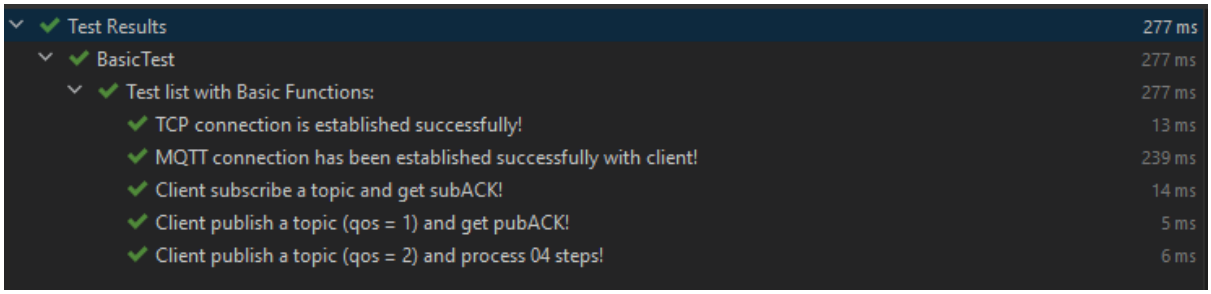


Integrated Functions

After checking all single functions in previous tests, this test is used to evaluate all the main functions when the whole system react to handle a request from clients.

The method is, clients are created and try to send several of MQTT requests such as: connecting, subscribing, publishing… and look for the reaction from server. In the case system can adapt all these requires, it is ready to go live.

The list of tests below shows the functions which are tested and qualified.



Conclusion: Regarding the result, all basic functions pass the tests.

To conclude, this way to save the effort for debugging, by testing single case and integrated case, it can easily detect the problems, where and how to fix them.

4.3.2. Integration Tests:

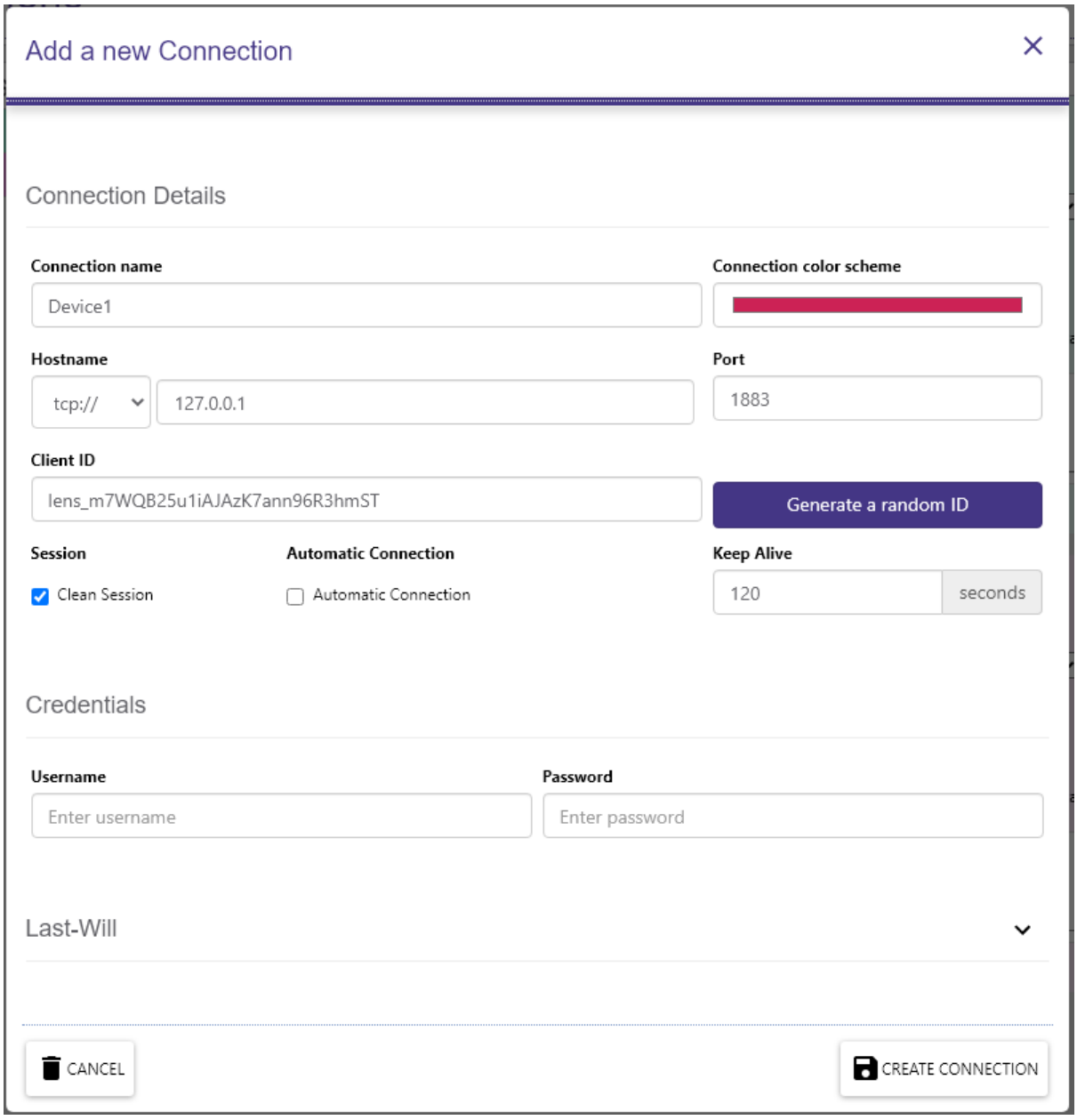
Tool test

However, we need to analyze the overall performance of the whole system, which should behave like a genuine MQTT Broker, thus an external tool will be utilized in this situation. This tool emulates MQTT clients that attempt to connect to and publish/subscribe to our Broker. MQTTLens, which is available in the Chrome Store, is utilized.

We will go through and evaluate all the functions. So here we go.

Connection Function

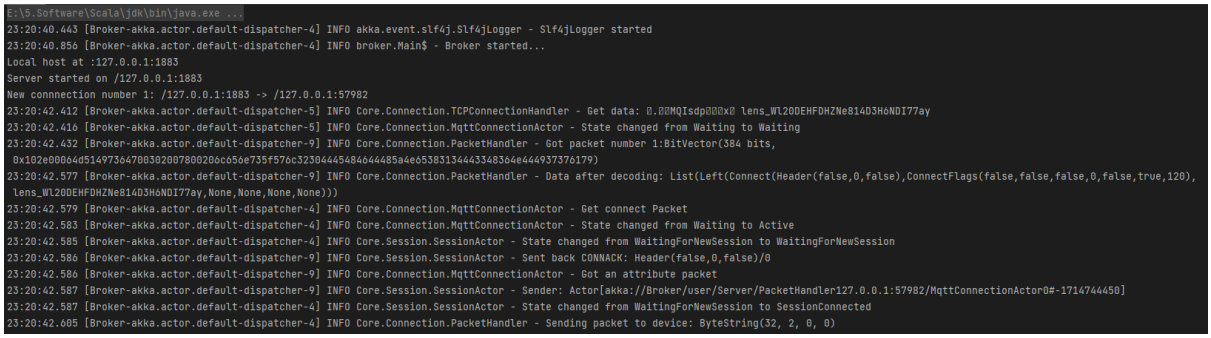
First of all, we need to create a new MQTT client (or a new device/connection) named "Device1". The image below shows how the information has to be set, including: ***Hostname***, ***Port***, ***Keep Alive*** (Timeout of connection), and ***Clean Session*** with MQTTlens.



After that, we create and connect Device1 to our Broker by pressing CREATE CONNECTION. On the other hand, from Broker side, the Connection Request has been received and proceeded. Finally, in the end of the logger, we see the line:

```
23:20:42.605 [Broker-akka.actor.default-dispatcher-4] INFO
Core.Connection.PacketHandler - Sending packet to device: ByteString(32, 2, 0, 0)
```

The packet **(32, 2, 0, 0)** is definitely the CONNACK packet, that means the MQTT connection in between them has been established.



Now, by repeating those previous steps, we will create 03 MQTT connections named: Device1, Device2 and Device3.

Publish/Subscribe Function and Retain Flag

Theory review: If a client subscribes to a topic, it will be updated from the Broker whenever that topic has been published with new packets. Furthermore, the retain flag is quite crucial. If this flag is set to 1 when publishing the packet, the Broker is required to preserve the packet in case a new client subscribes to the same topic, that client will get the latest packet with retain flag = 1. If the retain flag is set to 1 and the QoS is set to 0, it will remove all previously broadcast packets.

In this evaluation, Device1 serves as a Publisher, while Devices 2 and 3 serve as Subscribers. Then, as a scenario, these devices publish and subscribe in order to test the Broker's Functions. The following table details the actions of various devices:

Time	Device1 (Publisher)	Device2 (Subscriber)	Device3 (Subscriber)
11:39:01	Topic: topic1 Message send: “message1” Retain: False		
11:39:10		Topic: topic1 Message receive: None	
11:39:17	Topic: topic1 Message send: “message1” Retain: False	Topic: topic1 Message receive: “message1”	
11:39:32	Topic: topic1 Message send: “message2” Retain: True	Topic: topic1 Message receive: “message2”	
11:39:45	Topic: topic1 Message send: “message3” Retain: False	Topic: topic1 Message receive: “message3”	
11:39:51			Topic: topic1 Message receive: “message2”

Now we analyze the timeline:

11:39:01: Device1 published topic “topic1” with message “message1”, retain flag = false.

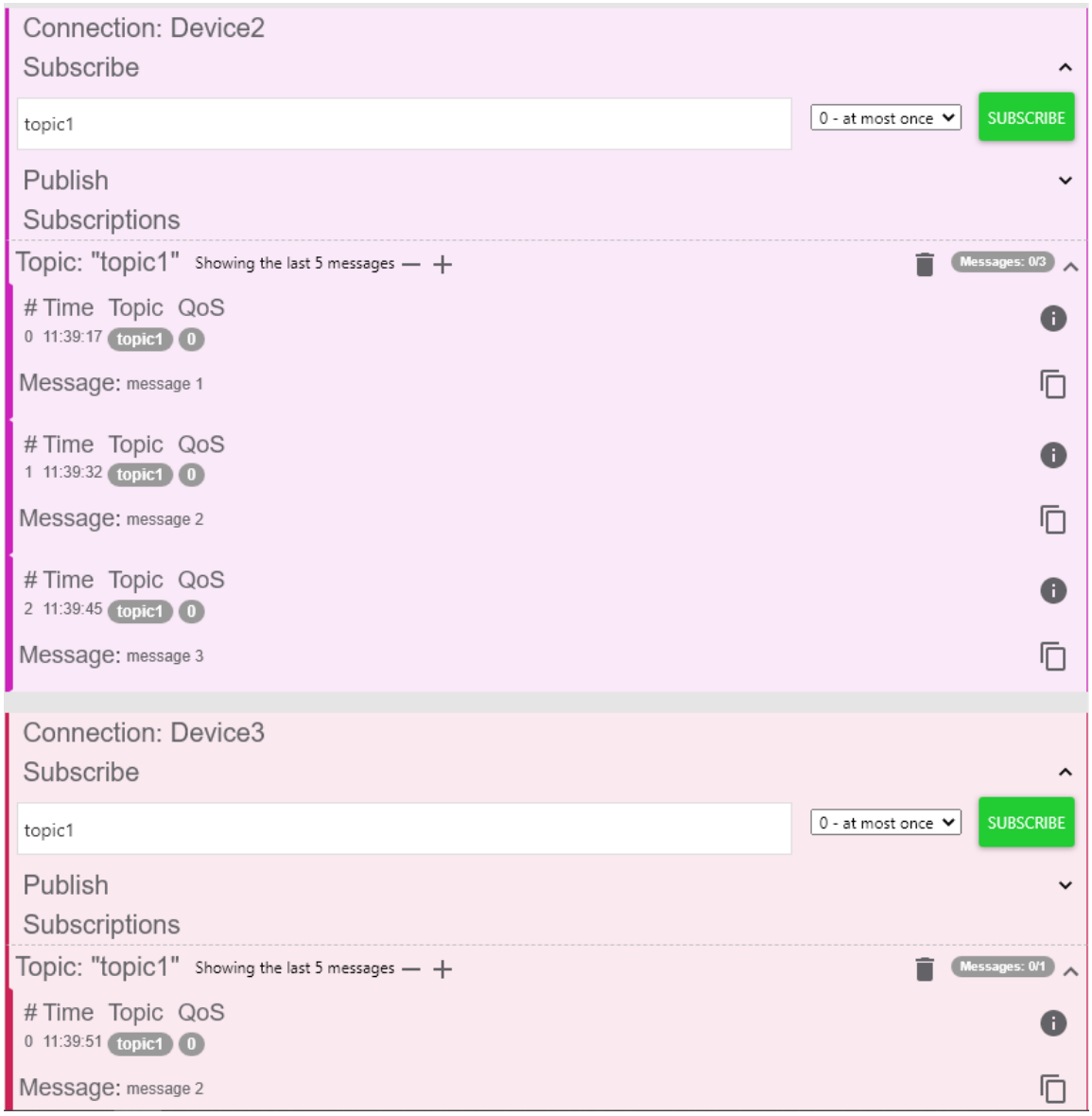
11:39:10: Device2 subscribed topic “topic1”, but received nothing, because the action subscribing happened after the action publishing from Device 1 and retain flag = false.

11:39:17: Device1 published topic “topic1” with message “message1”, retain flag = false. Meanwhile, Device2 received message “message1”, because Device2 already subscribed to that topic, ***that means the Publish and Subscribe functions work well in basic concept***.

11:39:32: Device1 published topic “topic1” with message “message2”, retain flag = true. At the same time, Device2 received message “message2”, again, ***the Publish and Subscribe functions work well in basic concept***. Note that retain flag now is true.

11:39:45: Device1 published topic “topic1” with message “message3”, retain flag = false. Since, Device2 received message “message3”, because Device2 already subscribed to that topic.

11:39:51: This time, Device3 joined, also subscribed “topic1” and it received message “message2”. Back to 11:39:10, Device2 subscribed “topic1” after Device1’s publishing but could not receive any message, but now Device3 could. The point here is, the retain flag was set to true at 11:39:32, so then Broker had to save that value (“message2”) and give it to any devices subscribe “topic2” since then. ***In short, that means the Publish and Subscribe functions work well with retain flag***.



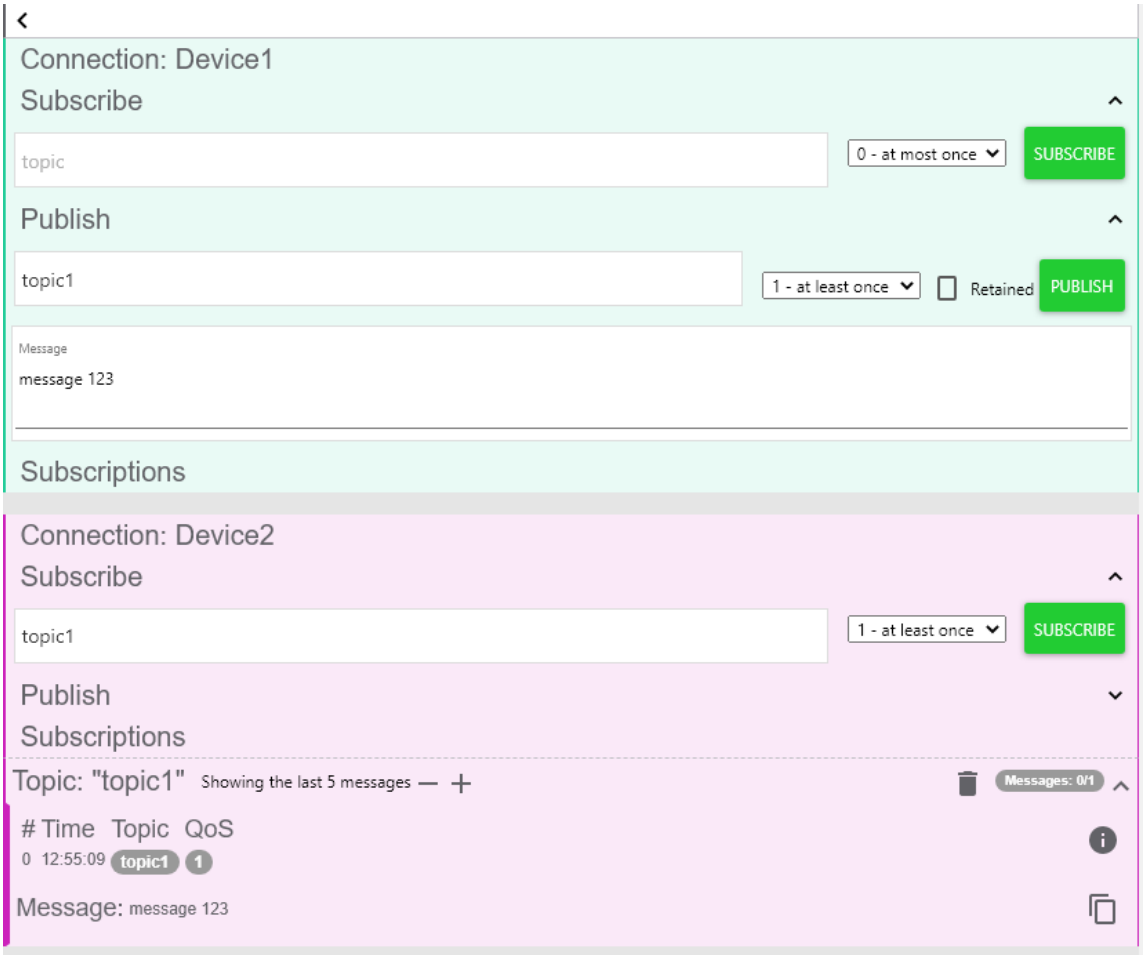
Qualities of service (QoS)

The Quality of Service (QoS) level is an agreement between a message's sender and recipient that establishes the assurance of delivery for a given message. MQTT has 03 degrees of QoS:

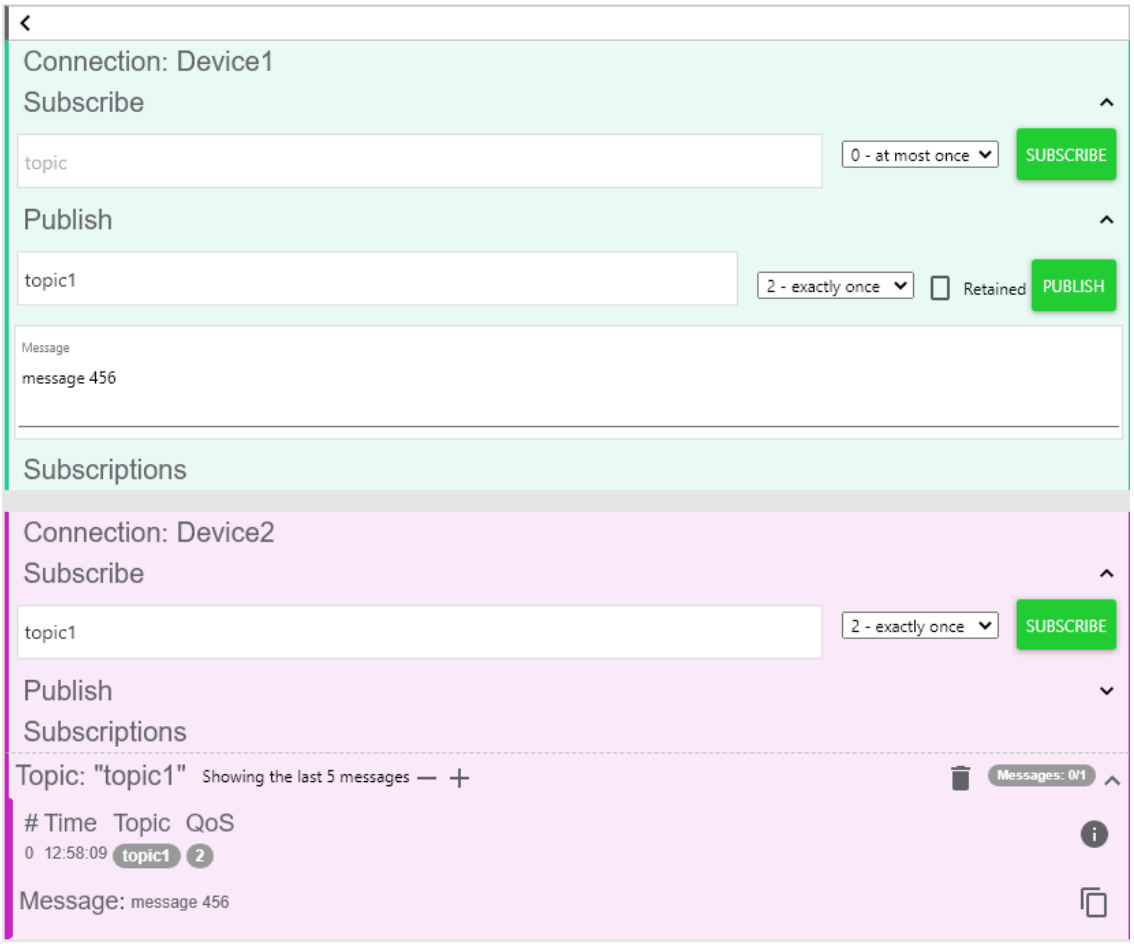
- ✓ ***At most once (0):*** QoS level 0 is sometimes known as "fire and forget" since it offers the same guarantees as the underlying TCP protocol.
- ✓ ***At least once (1):*** QoS level 1 ensures that a message is delivered to the receiver at least once. The message is stored by the sender until it receives a PUBACK packet from the receiver acknowledging receipt of the message. A communication has the potential to be transmitted or delivered several times.
- ✓ ***Exactly once (2):*** QoS 2 is the most secure and sluggish quality of service level. At least two request/response flows (a four-part handshake) between the sender and the recipient offer the assurance.

In this step, QoS compliant is verified by doing the simple test. Thanks to MQTTlens tool, the connection is supported to change the QoS. Then we test with QoS = 1 and QoS = 2 (QoS = 0) was tested in previous test.

QoS = 1:



QoS = 2:



As we can see, all the functions have been done correctly, that means the MQTT Broker Program work well in every single cases.

In another word, the task to implement MQTT Broker by using Scala and AKKA Model is done.

5. Developing of Dashboard:

5.1. The approach:

This part shows the idea for the project's whole system.

5.1.1. Model:

We basically solved more than a half of our finance demands after building the MQTT Broker application described above. Now, we have a MQTT Broker with full capability and power for the IoT system after deploying the MQTT Broker program on a server. However, it is only the inner core; what must be constructed in this thesis is a Dashboard, which is essentially a website that can directly interact and display with the program fragment.

To do this, we will try to build a web application like the architecture below:

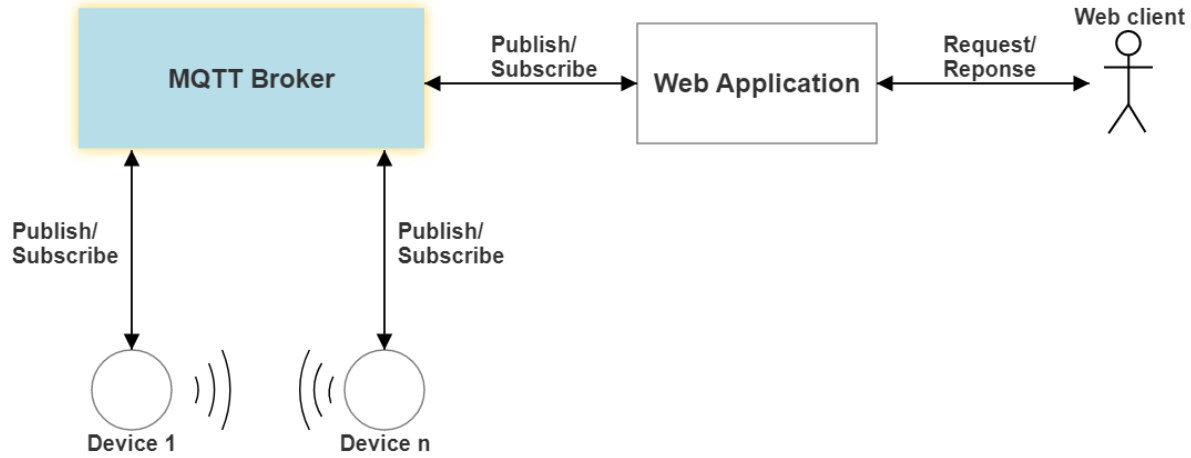


Figure 7. Illustrating of Project Architecture

The figure above depicts a view of the model of the user interface that will be built, which contains two main clients sides: one side includes devices and one side has user and web application. In the center is MQTT Broker.

The fundamental MQTT concept, the side from devices to Broker, is relatively recognizable. Devices in the IoT system will transmit MQTT packets (connect/publish/subscribe...) to the MQTT Broker's host address and receive corresponding response. On the other side, users want to monitor or send information (setup, control) to these devices through a *web client* application.

This web client points directly to the address of the server running the MQTT Broker program via an web application and will communicate with this Server using the HTTP protocol. The problem is, a web client can only communicate using HTTP request/response, while MQTT is a completely different protocol (although both are built on top of TCP/IP). So then, the Web Application needs to convert the HTTP requests to corresponding MQTT requests and send to Broker. On the other hand, the results returned in the form of MQTT packets will be converted back to HTTP and become the HTTP response of the web client's request.

By using the above model, users have in hand a Dashboard interface to directly interact with MQTT Broker, through which can create new clients, send information to clients and view desired information from them.

5.1.2. Security:

Security is a major concern in the network protocol, and the project employs two protocols: MQTT and HTTP. As a result, we must first understand about the security of various protocols, as well as how libraries and frameworks may assist us.

MQTT protocol

For beginning, with MQTT protocol, it has two security levels:

- First, MQTT permits a *userID* and *password* (or *clientID* – in some cases) to be communicated at the time of connection, and Broker allows authorization to be based on those. [16] This layer of security is implemented by Broker, by setting the rule of *clientID* or *userID*, or by restricting access to topics, such as control which clients are able to subscribe and publish to topics, Broker can prevent any attacks.
- Second, all versions of TLS up to and including 1.2 are supported for securing the underlying TCP/IP connection. This implies that MQTT connections may be encrypted, and the identities of connected apps can be confirmed in a more secure manner than using a *clientID* and *password*. [16] However, this method has one problem, that it requires client support, and it is unlikely to available on simple clients.

These methods ensure our MQTT network is protected from attackers. Besides, there are some other user defined methods, such as Payload Encryption. However, it is not popular and useful most of cases.

In case Paho Java Client – the library is used in this project (will be mentioned later), it supports fully TLS in their APIs, which is encrypting the communication between web applications and servers. TLS encryption can aid in the protection of online applications against data breaches and other types of assaults. [17]

However, because of no security support from our Broker Program, hence we just apply the normal MQTT protocol without TLS, and put it to improvements list in the future.

HTTPS protocol

On the other hand, we have HTTPS as a far more secure version of the HTTP protocol. Luckily, Play framework supports HTTPS and it is used in this project, for communicating between web client and web app.

HTTPS (Hypertext transfer protocol secure) is well-known as the secure version of HTTP, which is the primary protocol used to transfer information between a web browser and a website. HTTPS is encrypted in order to increase security of data transfer. [18]

To encrypt communications, HTTPS employs an encryption mechanism. Transport Layer Security is used (TLS). This protocol protects communications by utilizing an asymmetric public key infrastructure. This sort of security mechanism encrypts communications between two parties using two separate keys:

- ***The private key:*** This key is held by the owner of a website and, as the reader may have guessed, it is kept confidential. This key is stored on a web server and is used to decode data encrypted with the public key.
- ***The public key:*** This key is available to anybody who wishes to communicate with the server in a secure manner. Only the private key may decode information encrypted with the public key.

HTTPS prohibits websites from broadcasting their information in a way that anyone spying on the network may readily access. When data is transferred via standard HTTP, it is divided into packets of data that may be readily "sniffed" using free software. HTTPS encrypts traffic

so that even if packets are sniffed or otherwise intercepted, they appear as gibberish characters. [18]

5.2. Software design of programs:

Regarding previous part, the main idea for approaching the challenge of constructing the Dashboard interface is to build a web client that interacts with users, then convert those interactions into HTTP requests to send to the Server. The server's answer (HTTP response) will be processed and shown to the user.

In this part, we use Scala and Play framework to develop the website.

5.2.1. Conceptions:

Play Framework:
Play is a high-productivity Java and Scala web application framework that unifies components and APIs for contemporary online application development. Play was created by web developers for the creation of online applications. [19]

Play is a full-stack framework, base on Model-View-Controller (MVC) architecture, it provides everything you need to develop Web Applications and REST services, including an integrated HTTP server, form handling, CSRF protection, a robust routing system, support, and more.

Under the hood, Play's lightweight, stateless, web-friendly design employs Akka and Akka Streams to guarantee predictable and low resource use (CPU, memory, threads). Applications scale organically, both horizontally and vertically, thanks to its reactive approach. [19]

MVC Architecture:
Model-view-controller (MVC) is a software architecture paradigm often used for building user interfaces that divides the underlying program logic into three interrelated pieces. This is done to separate internal representations of information from how it is presented to and accepted by the user. [20]

- ✓ **Model:** The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. [20] It directly manages the data, logic and rules of the application.
- ✓ **View:** Any information representation, such as a chart, diagram, or table. Multiple perspectives of the same information, such as a bar chart for management and a tabular view for accountants, are feasible.
- ✓ **Controller:** Accepts input and converts it to commands for the model or view. [21]

5.2.2. Architecture of a general Play project:

Base on MVC achitecture, a Play project structure looks like this:

```
app                   → Application sources
 └ assets             → Compiled asset sources
   └ stylesheets      → Typically LESS CSS sources
   └ javascripts      → Typically CoffeeScript sources
 └ controllers        → Application controllers
 └ models             → Application business layer
 └ views              → Templates
build.sbt             → Application build script
conf                  → Configurations files and other non-compiled resources (on classpath)
 └ application.conf   → Main configuration file
 └ routes             → Routes definition
dist                  → Arbitrary files to be included in your projects distribution
public                → Public assets
 └ stylesheets        → CSS files
 └ javascripts        → Javascript files
 └ images             → Image files
project               → sbt configuration files
 └ build.properties           → Marker for sbt project
 └ plugins.sbt                → sbt plugins including the declaration for Play itself
lib                   → Unmanaged libraries dependencies
logs                  → Logs folder
 └ application.log    → Default log file
target                → Generated stuff
 └ resolution-cache   → Info about dependencies
 └ scala-2.11
   └ api              → Generated API docs
   └ classes          → Compiled class files
   └ routes           → Sources generated from routes
   └ twirl            → Sources generated from templates
 └ universal          → Application packaging
 └ web                → Compiled web assets
test                  → source folder for unit or functional tests
```

The app/directory

The app directory includes all executable artifacts, including Java and Scala source code, templates, and the sources of built assets.

In the *app* directory, three packages which represent for each component of the MVC architectural pattern:

- *app/controllers*
- *app/models*
- *app/views*

Users can add-in their own packages, for instance, an *app/utils* package.

The public/directory

The static assets saved in the *public* directory are provided directly by the Web server.

There are three sub-directories for images, CSS stylesheets and JavaScript files in this directory. To keep all Play applications consistent, developers should organize static assets as default like this.

The conf/directory
The configuration files for the program are stored in the *conf* directory. The major configuration files are::

- *application.conf*, the main configuration file for the application, which contains configuration parameters
- *routes*, the routes definition file.

It's a good idea to add extra options to the *application.conf* file if you need to add configuration settings particular to your application.

Additionally, if a library needs a specific configuration file, put it under the *conf* directory.

The lib/directory
The *lib* directory is optional, and it contains unmanaged library dependencies, such as all JAR files that you want to handle manually outside of the build system. Simply place any JAR files here and they will be added to the class path of your application.

The build.sbt file
The main build declarations of the project are generally found in *build.sbt*, which places at the root of the project.

The project/ directory
The *project* directory contains the sbt build definitions:

- *plugins.sbt* defines sbt plugins used by this project.
- *build.properties* contains the sbt version to use to build the app.

The target/directory
The *target* directory contains everything generated by the build system. The list below show which are generated:

- *classes/* includes all compiled classes (from sources code).
- *classes_managed/* contains only the classes that are managed by the framework (such as the classes generated by the router or the template system). It can be useful to add this class folder as an external class folder in your IDE project.
- *resource_managed/* contains generated resources, typically compiled assets such as LESS CSS and CoffeeScript compilation results.
- *src_managed/* contains generated sources, such as the Scala sources generated by the template system.
- *web/* contains assets processed by sbt-web such as those from the *app/assets* and *public* folders.

Typical .gitignore file
Generated folders should be ignored by your version control system.

5.3. Implementation:

5.3.1. Objects:

This step, a web application will be developed to:

- Register new devices, config the parameters
- Connect/send/receive MQTT packets with MQTT Broker
- Visualize data on dashboard

The user interface will be explained clearer in the next part, this part we just try to implement the logic inside of the web application, or in another word, the backend of the website.

In part 5.2, structure of a Play project was showed and explained, now we use that template to develop our website. That means the three main packages and other directories already have been created and we just base on that to add some user functions to help implement the objects above.

5.3.2. Implementation:

It would be redundant to reveal all of the code in this report; instead, the author will only describe the website's operation and discuss some of the programs/techniques used to construct the functionality and web interface.

To view full source code, please access the link github repository of project [14].

Homepage
Firstly, we need a home page to serve as the initial interface for all subsequent phases. On this interface page, we list the devices together with their *descriptions*, *clientIDs*, etc. Simultaneously, there will be function buttons for adding new devices and interacting with previously established gadgets.

The list of devices is loaded from the database; dealing with the database will be discussed in more depth later, but we understand that each device will be completely exposed to the capabilities of a MQTT Client, including: *Exploring*, as well as *Editing* and *Deleting*.

By default, the website runs on *http://localhost:9000*

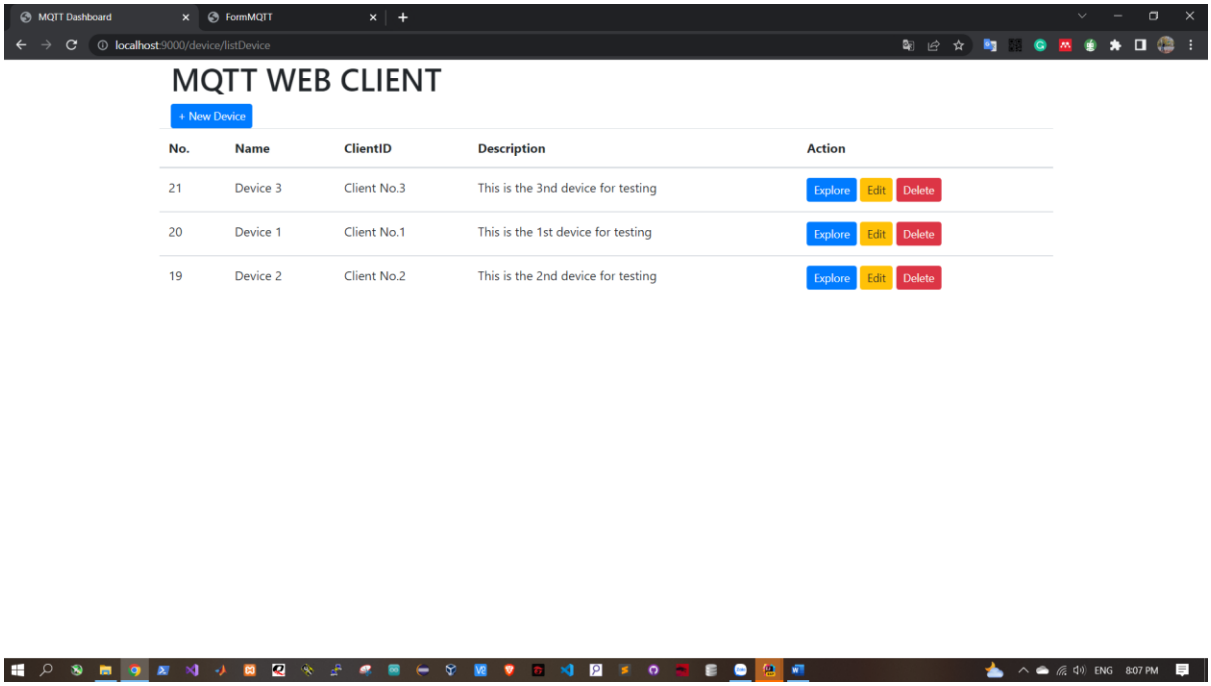


Figure 8. Homepage of Dashboard

Register for new devices, config the parameters

In this part, we need to deal with database. The reason is, any registered devices need to be memorized and stored in the database.

Thanks to Slick, which is is Lightbend's Functional Relational Mapping (FRM) library for Scala that makes it easy to work with relational databases. It allows you to interact with stored data virtually as if you were using Scala collections, while also allowing you complete control over when database access occurs and what data is sent. In Play framework, you can use SQL directly, but when by using Scala for your queries instead of plain SQL, you have compile-time safety and compositionality.

So let's start with Slick and registering devices functions.

At first, SQLite is used to manage the database, the reason is that is lightweight and does not require any complicated settings that affect the structure of the entire project. In the SQLite application, a device table is created with the following fields: *id*, *name*, *description*, *device_code (clientID)*, and points to the database file produced for the website. From now on, we work with this table, and the list of devices will be persistently and precisely kept each time the site is launched.

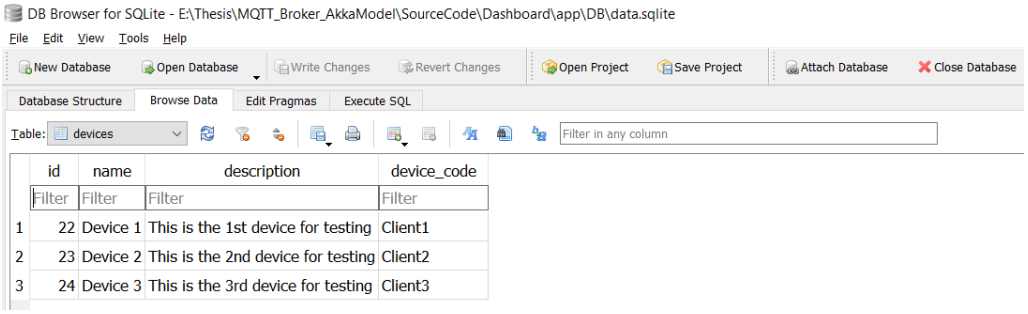


Figure 9. Setup devices table (in database)

Next, a form is developed to fill the information of device that needs to be registered. In our project, devices have their own *name*, *code* and *description*, so then we build a form which help to type and submit *Device Name*, *Device ID* and *Description*. Note that all forms are using POST method to secure the request.

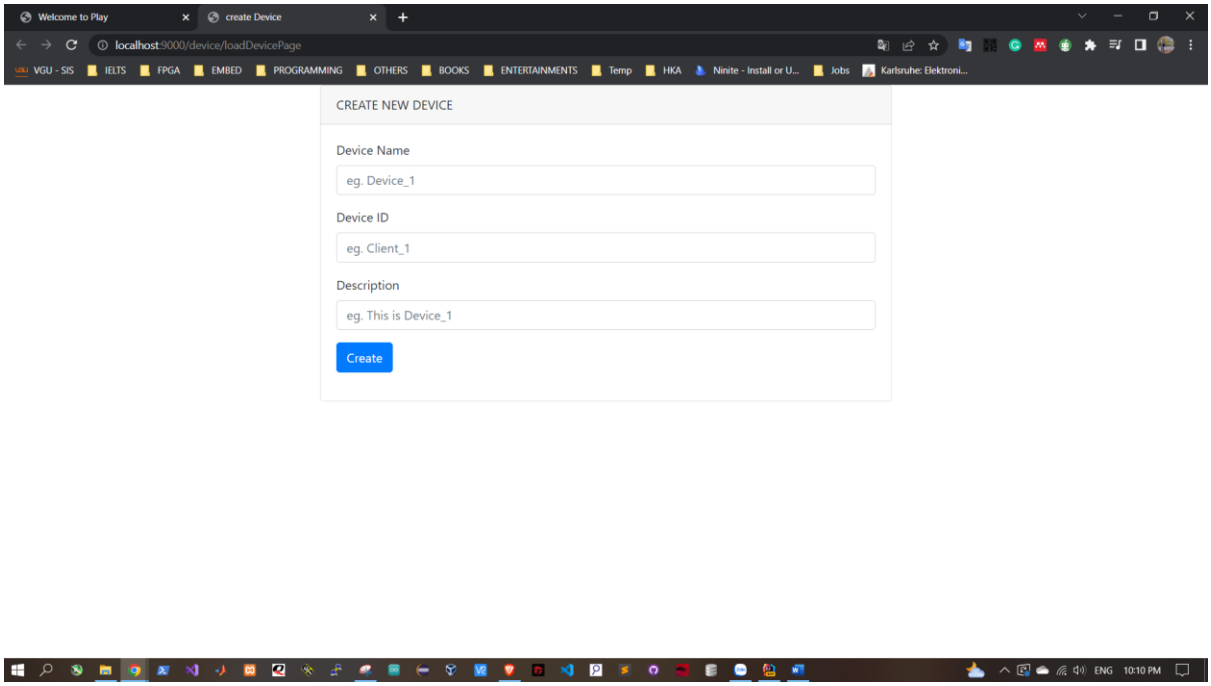


Figure 10. Web UI for creating device

After submitting the request, from back-end view, a handler is developed to process the request. This function follows the flowchart below:

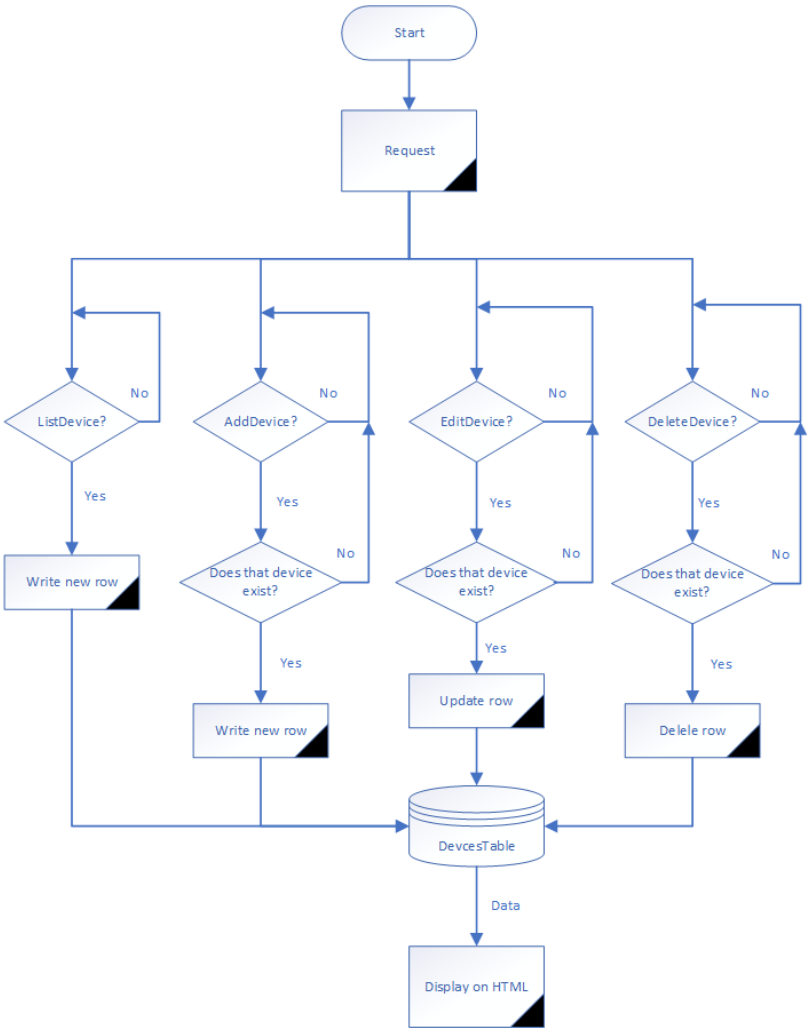


Figure 11. Flowchart for database excution

Next, we have to deal with model, which manages and process the data. In this case, the model uses to execute three main functions:

- ***Create new device***: Check if the device has not been declared and does not exist on the system, build a new data store to contain the device information.
- ***Delete device***: Check for the existence of the device and erase the device name from the data space where the device information is saved.
- ***Edit device***: Check to see if the device exists, and if so, modify the device's details.

To do that, a class of device table is created in the Model package:

```
case class Device(id: Option[Int], name: String, device_code: String, description:
String)
// Definition of the SUPPLIERS table
class Devices(tag: Tag) extends Table[Device](tag, "devices") {
  def id = column[Int]("id", O.PrimaryKey, O.AutoInc) // This is the primary key
column
  def name = column[String]("name")
  def description = column[String]("description")
  def device_code = column[String]("device_code", O.PrimaryKey)
  def * = (id.?, name, device_code, description) <> (Device.tupled, Device.unapply)
  // Every table needs a * projection with the same type as the table's type
parameter
```

```
//  def * = (id.?, name, device_code, description)
}
```

Then, to implement the tasks with database, the code block below is used:

For adding new device:

```
def addItem(device_name: String, device_code: String, device_description:
String): Unit = {
  val query = db.run(devicesTable.filter(i => i.device_code === device_code ||
i.name === device_name).exists.result)
  val check = Await.result(query, 5 seconds).asInstanceOf[Boolean]
  if (!check) {
    val addNewDevice = (devicesTable returning devicesTable.map(_.id)) +=
Device(None, device_name, device_code, device_description)
    val test = Await.result(db.run(addNewDevice), 20 seconds)
  }
}
```

For editing device's information:

```
def editItem(device_code: String, nName: String, nCode: String, nDes:
String): Unit = {
    val query = devicesTable.filter(_.device_code === device_code)
      .map(devicesTable=>(devicesTable.name, devicesTable.device_code,
devicesTable.description)).update(nName, nCode, nDes)
//    val editDevice = query.update(nName, nCode, nDes)
    val update = Await.result(db.run(query), 10 seconds)
  }
```

For deleting device:

```
def deleteItem(device_code: String): Unit = {
  val query = devicesTable.filter(_.device_code === device_code)
  val deleteDevice = query.delete
  val delete = Await.result(db.run(deleteDevice), 10 seconds)
}
```

Finally, thanks to Slick, the library both supports SQL queries and supports Scala's great feature of asynchronous, which makes querying data accessible and without any blocking the program.

Connect/send/receive MQTT packets with MQTT Broker

Assuming that the device was successfully constructed, it must have some functionalities in order to serve as a client in the system. Connecting to Broker, subscribing topic to Broker to take data, and publishing data from its side are the major functions.

To begin, we need a form that allows users to interact with devices and request for information via MQTT protocol.

The form below is built to satisfy that kind of requirements, in the simplest UI.

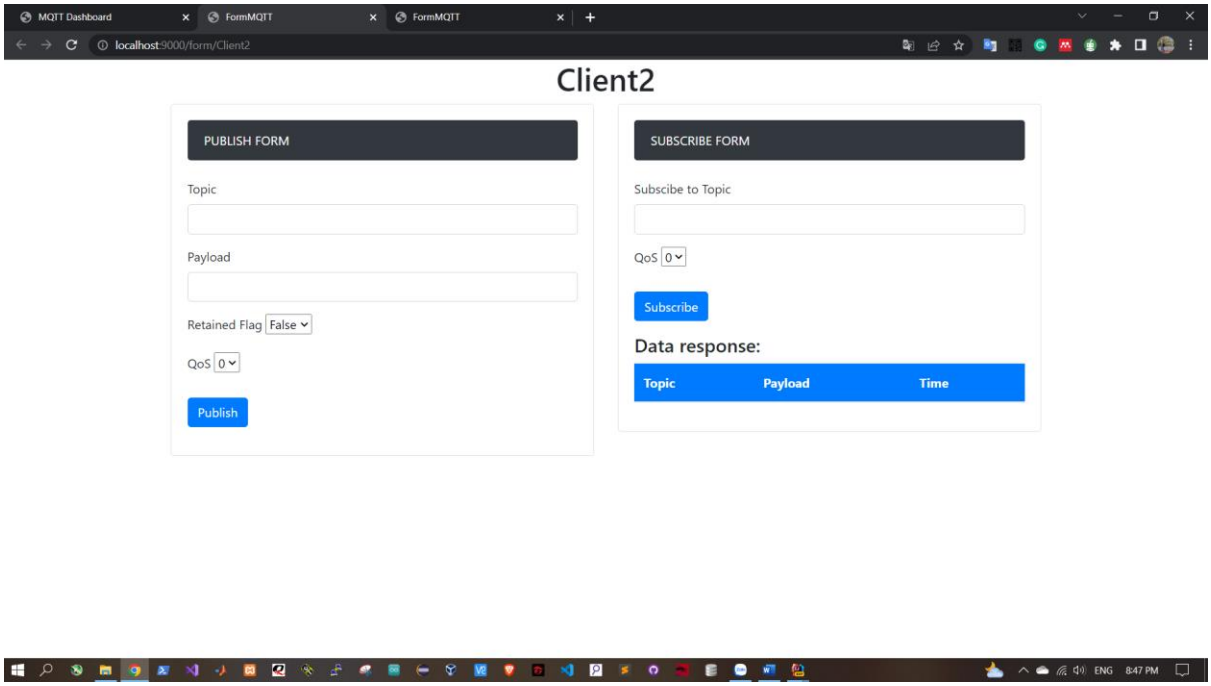


Figure 12. Web UI for implementing MQTT communication

In this integrated form, there are two separated form such as: Publish Form and Subscribe From. In case of Publish Form, there are boxes for typing topics and payload (to publish), and drop list for choosing Retained Flag status and QoS as well. On the other hand, Subscribe From has only one box for typing topic (to subscribe) and drop list for QoS. In general, these two form serve full main basic parameters and functions for users to implement a MQTT communication.

Sencondly, moving to implement the execution of MQTT protocol, we use a library – which is provided by Paho.

The Paho Java Client is a Java-based MQTT client library for designing apps that operate on the JVM or other Java-compatible platforms.

The Paho Java Client comes with two APIs: *MqttAsyncClient* provides a completely asychronous API in which activity completion is communicated via registered callbacks. *MqttClient* is a synchronous wrapper for *MqttAsyncClient* in which functions appear to the application synchronously.

The reason for choosing this library is to save the time to build up a new program for MQTT client functions as: connecting/subscribing/publishing… which are effectively handled. By intergrating this library with backend, the website can handle the MQTT communication with the Broker.

To use mqtt library, we add on the dependency as below to sbt file:

```
libraryDependencies ++= Seq(
"org.eclipse.paho" % "org.eclipse.paho.client.mqttv3" % Mqttv3Version,
```

Then we implement the logic of MQTT protocol by coding:

```
package services

import models.PayloadItem
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence
import org.eclipse.paho.client.mqttv3.{IMqttDeliveryToken, MqttCallback,
MqttClient, MqttMessage}

import javax.inject.{Inject, Singleton}
import scala.util.{Failure, Success, Try}

/**
 * MQTT Services
 * This class supports all functions of MQTT protocol, which is developed
base on MQTT Paho Client Library
 * Including: Subscribe/Publish/Connect
 */
object BrokerInfo{
  val url = "127.0.0.1"
  val port = 1883
}
@Singleton
class MQTTServices @Inject() (clientID: String) {

  //setting up to connect to Broker
  val brokerUrl = s"tcp://${BrokerInfo.url}:${BrokerInfo.port}"
  val persistence = new MemoryPersistence
  val client = new MqttClient(brokerUrl, clientID, persistence)
  var payloadsList: List[PayloadItem] = List()

  //subscribe function
  def subscribe(topic: String, qos: Int ): Unit = {
    //check if the device already connected to broker
    if (!client.isConnected()){
      client.connect()
      println(s"$clientID connecting to $brokerUrl ...")
    }

    client.subscribe(topic, qos)
    println(s"$clientID suscribed to $brokerUrl with topic $topic ...")

    val callback = new MqttCallback {
      override def deliveryComplete(token: IMqttDeliveryToken): Unit = {
        println("Delivery complete!")
      }

      override def connectionLost(cause: Throwable): Unit = {
        println("Connection to socket lost")
      }

      override def messageArrived(topic: String, message: MqttMessage):
Unit = {
        val newPayload = new String(message.getPayload)
        println(s"SUBSCRIBE | $clientID | received message: $newPayload -
TOPIC: $topic")
        payloadsList = new PayloadItem(clientID, topic, newPayload) ::
payloadsList
      }
    }
    client.setCallback(callback)
  }
```

```
  def getUpdate: List[PayloadItem] = {

    payloadsList
  }

  //publish function
  def publish(topic: String, message: String, retain_flag: Boolean, qos:
Int): Boolean = {
    //check if the device already connected to broker
    if (!client.isConnected()) {
      client.connect()
      println(s"CONNECT | $clientID | connect to broker: $brokerUrl ...")
    }

    val result = Try(client.getTopic(topic)) match {
      case Success(messageTopic) =>
        val mqttMessage = new MqttMessage(message.getBytes("utf-8"))
        mqttMessage.setRetained(retain_flag)
        mqttMessage.setQos(qos)

        Try(messageTopic.publish(mqttMessage)) match {
          case Success(r) =>
            println(s"PUBLISH | $clientID | sent message: ${r.getMessage}")
            true
          case Failure(exception) =>

            println(
              s"""
                 |An error occurred while trying to publish the message
"$message" to $topic into $brokerUrl:
          """.stripMargin, exception)
            false
        }
      case Failure(exception) =>
        println(
          s"""
             |An error occurred while trying to publish the message
"$message" to $topic into $brokerUrl:
          """.stripMargin, exception)
        false
    }
    result
  }
}
```

In order to conduct connection, publish, and subscribe operations for the device, a MQTTServices Class is developed. Therefore, a MQTTServices object will be generated for every new device that is formed and communicates with the broker to handle all of that device's MQTT communications, which are managed by the device's clientID.

Passing parameters obtained from HTML format such as themes, payload, clientID, QoS etc. and delivering to Broker using Paho library makes publishing data reasonably simpleSubscribing data, on the other hand, is more complicated since it is essential to establish a connection with the Broker, maintain that connection, and listen to all events from the Broker, which necessitates the usage of an asynchronous function in the Broker. Paho is a mqtt Callback library. A Callback will be configured to listen for Broker events and run routines to handle such events. When a new message is sent to a subscribed topic, for example, the device will get it without any additional trigger.

Visualizing Data

Finally, after obtaining data on the backend, such data, particularly subscription data, must be disclosed to the user.

The issue is that loading the entire page would be a waste of time and resources, thus in this situation, we utilize AJAX so that just the portion of the website holding the payload would get events from the Broker changed and shown to the user after being received from the Broker.

AJAX = **A**synchronous **J**avaScript **A**nd **X**ML.

AJAX is just a combination of:

- ✓ An XMLHttpRequest object integrated into the browser (to request data from a web server)
- ✓ HTML DOM and JavaScript (to display or use the data)

AJAX enables asynchronous updating of web pages by secretly sharing data with a web server. This indicates that a web page may be updated in sections without requiring a page reload. [22]

This is the result, the table of payloads (Data response) is updated continously without any webpage reloading.

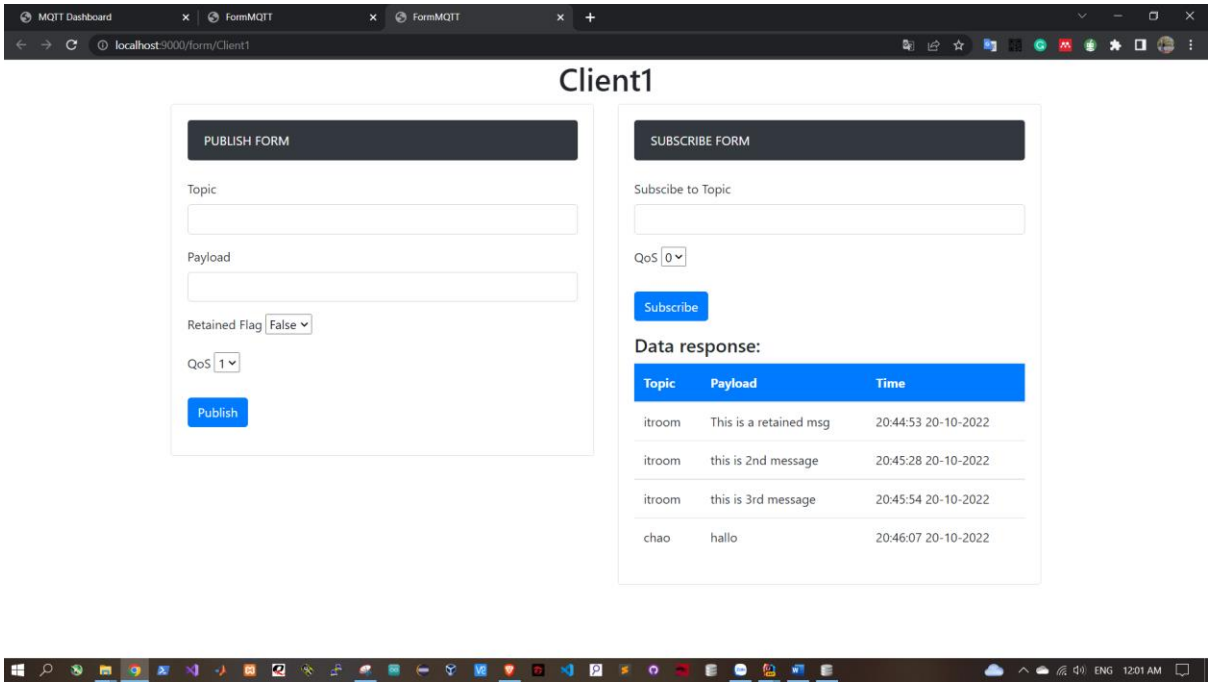


Figure 13. Payload is visualized in Website

5.4. Testing and Result:

After developing the website, the last and very important thing, is testing. We use the Test Kit which is provided in Play Framework.

Test with Browser

In this test, we try to bind the website on localhost and port 9000, to make sure the website can run normally on a browser enviroment.

```
[info] - should work from within a browser
[info] Run completed in 6 seconds, 651 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 10 s, completed Oct 21, 2022, 8:44:36 PM
[IJ][Dashboard] $
[IJ][Dashboard] $
```

Result: Passed!

Routing Test

This test is used to check how the website react with good and bad requests.

```
[info] Routes
[info] - should send 404 on a bad request
[info] - should send 200 on a good request
[info] - should send 303 on a good request but redirect to another site
[info] Run completed in 3 seconds, 464 milliseconds.
[info] Total number of tests run: 3
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 3, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 5 s, completed Oct 21, 2022, 9:51:39 PM
[IJ][Dashboard] $
[IJ][Dashboard] $
```

Result: All tests passed!

Rendering Test

The test is applied to make sure the website can render all the html pages which are defined in the program.

```
[info] HomeController
[info] - should render the HOME PAGE successfully!
[info] - should render the PUBSUB FORM successfully!
[info] - should render the CREATE DEVICE FORM successfully!
[info] - should render the EDIT DEVICE FORM successfully!
List(models.DeviceItem@1a454a8, models.DeviceItem@789cfc56, models.DeviceItem@11b80793)
[info] - should render the LIST DEVICE PAGE successfully!
[info] Run completed in 4 seconds, 152 milliseconds.
[info] Total number of tests run: 5
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 5, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 6 s, completed Oct 21, 2022, 10:05:08 PM
[IJ][Dashboard] $
[IJ][Dashboard] $
```

Result: All html pages are rendered successfully!

Services Test

For this test, we try to implement all functions of MQTT and other actions we have to deal with the website such as: publishing, subscribing… These tests are to demonstrate the website can excute the requests from users.

```
 {file:/E:/Thesis/MQTT_Broker_AkkaModel/SourceCode/Dashboard/}root/ testOnly UnitSpec
[info] UnitSpec:
[info] MQTTServices
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#noProviders for further details.
SLF4J: Class path contains SLF4J bindings targeting slf4j-api versions prior to 1.8.
SLF4J: Ignoring binding found at [jar:file:/C:/Users/USER/AppData/Local/Coursier/cache/v1/https/
 .2.11.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#ignoredBindings for an explanation.
CONNECT | Client1 | connect to broker: tcp://127.0.0.1:1883 ...
PUBLISH | Client1 | sent message: test_msg
[info] - should CONNECT and PUBLISH a message successfully
Client1 connecting to tcp://127.0.0.1:1883 ...
Client1 suscribed to tcp://127.0.0.1:1883 with topic test_topic ...
SUBSCRIBE | Client1 | received message: test_msg - TOPIC: test_topic
[info] - should CONNECT and SUBSCRIBE a message successfully
[info] Run completed in 3 seconds, 386 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 5 s, completed Oct 21, 2022, 11:34:27 PM
[IJ][Dashboard] $
[IJ][Dashboard] $
```

Result: All tests passed!

Database Test

This test is to check the ability to deal with database in this project.

```
[info] DatabaseTest:
[info] Database
[info] - should be creatable
[info] - should be accessible
[info] - should be editable
[info] - should be removable
[info] Run completed in 1 second, 839 milliseconds.
[info] Total number of tests run: 4
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 4, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 4 s, completed Oct 22, 2022, 12:14:05 AM
[IJ][Dashboard] $
[IJ][Dashboard] $
```

Result: All requests to databased are excuted successfully!

To conclude, any separated functions of website are sastified the requirements.

6. Integration of MQTT Dashboard:

After many steps, finally the Dashboard has been built. Actually, the dashboard contains two main programs: the Core Broker program – which provides the Broker's services of the system and the Web Client – which is the View for user to interact with the Broker.

By lauching them together, a MQTT Dashboard is running!

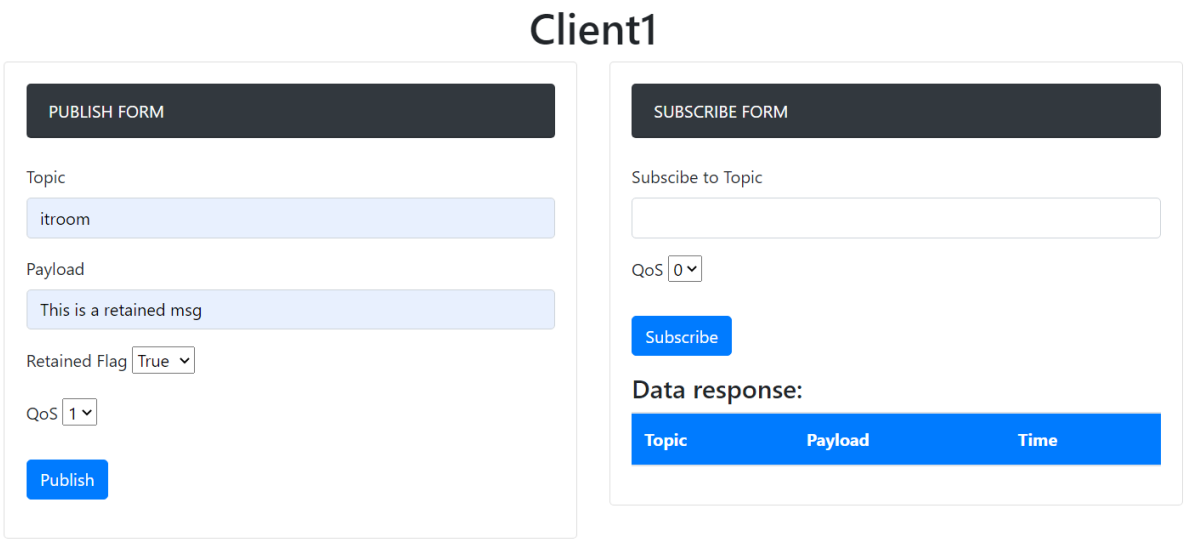
6.1. Integrating and lauching:

In this part, we run these two programs like a real application and observe their reaction for any single cases.

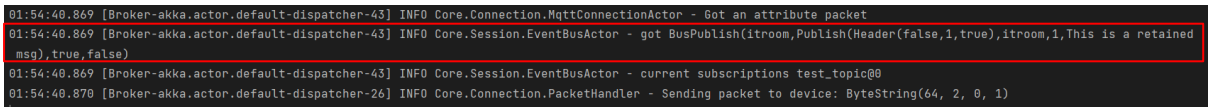
The Broker program will be launched on localhost (and maybe a public address if feasible) with an open port to listen and deliver MQTT services, followed by the Web Client, which will refer straight to the Broker's address and give us with the appropriate interfaces.

These show cases perform the most critical scenarios:

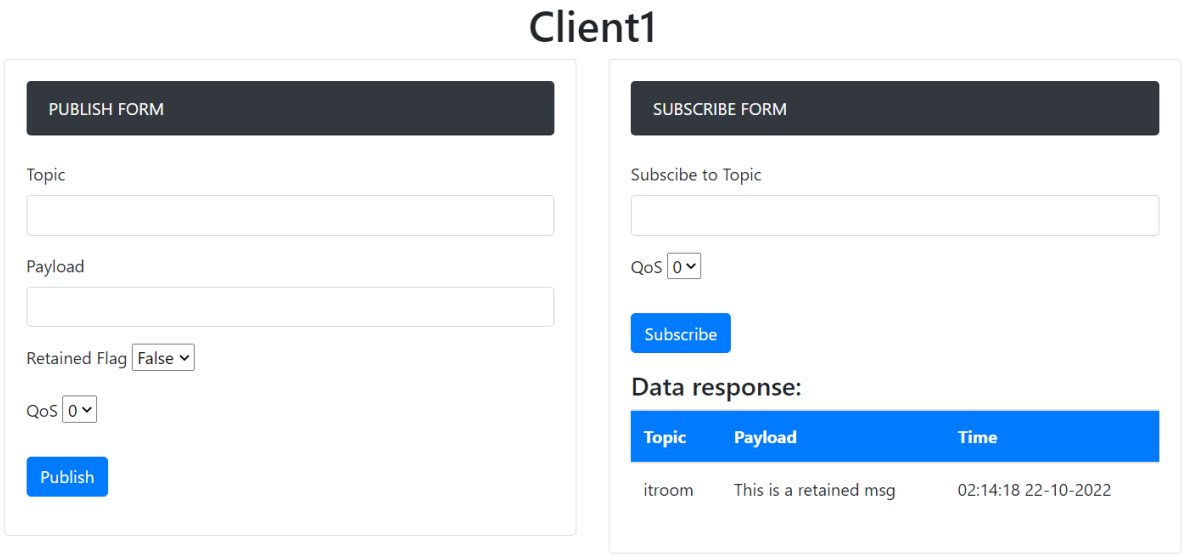
Publish with Retain Flag
In the MQTT form, a message is published with Retain Flag set to true.



From Broker side, we observe a Publish packet with correct information, that means this function work well from both sides.



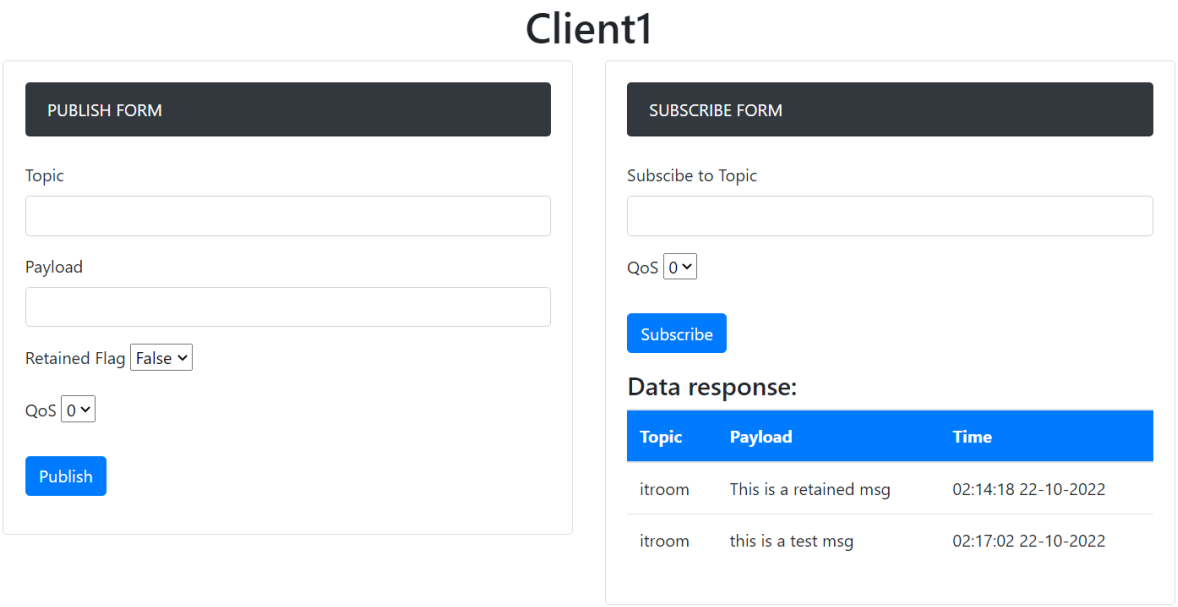
Next, a client subscribe exactly to that topic and receive that message.



Then, this function is success.

Publish without Retain Flag

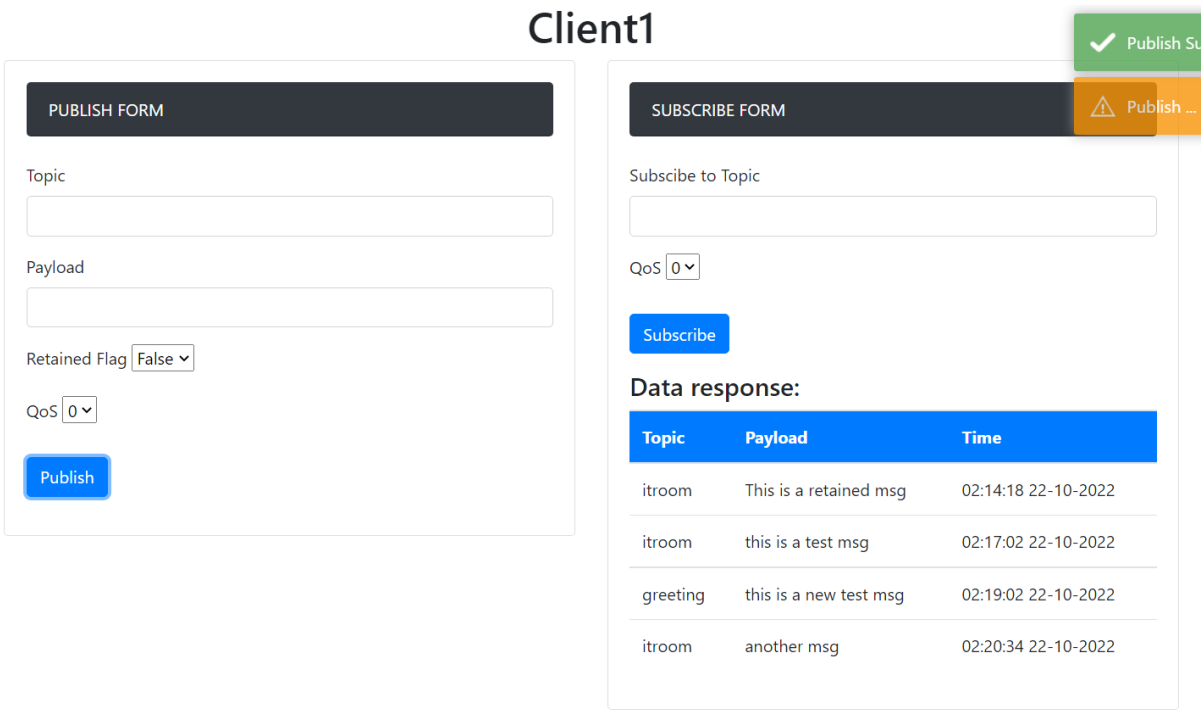
In this implementation, another message is publish without Retain Flag, but because the topic is already subscribe, the client also receives new message.



Subscribe multiple topics

Client subscribes another topic (greeting), and receives any new message from these topics (the new and the old one).

Number of subscribed topics is not limited.



For futher demonstration, please access the clip on [my github] to observe how the dashboard works in detail.

6.2. Conclusion:

Based on the favorable outcomes in Section 6.1, we can determine that the project completely integrated the elements that were suggested from the beginning.

Actually, developing a MQTT Broker and a MQTT Dashboard is not a new topic, numerous organizations, corporations, and people have developed these to a high level of perfection and sophisticationThe Actor Model and the AKKA framework are utilized to develop this application, which makes it unique. Currently, according to internet search results, this route (using Actor Model and AKKA framework) is not common, and nearly no one has created a MQTT model using these tools.

Through this, I also hope that this thesis will lead the way for future research to further develop the aspects will be described in Part 7 of this thesis.

7. Ideas for Improvements:

Aside from what has been accomplished, this project contains limits that can be addressed in the future.

7.1. Security:

Unfortunately, the development of security features was not implemented in this project, so adding security features is necessary and a top priority to improve the quality of the project.

MQTT Broker

- ✓ Implementing TLS (Transport Layer Security) from Broker side can help the requests to be encryted, then improve the security in communicaton in the system.
- ✓ In addition, it is possible to set some unique identification features for the system such as user, password, and user-defined information to determine the legitimacy of the devices that want to log in or operate in the network.

MQTT Client

- ✓ Using HTTPS in http requests from web client to web application to improve the security.
- ✓ Adding TLS to encryt all messages sent from MQTT client.
- ✓ Adding user-defined methods to cross check in the system.

7.2. More user functions:

In terms of user experience, the project is not sastified. Basically, it just provides the idea to implement the logics of MQTT in very simple form, so any improvement about UI, UX and additional functions are encouraged. The author can list here some features can be developed to gain user experience:

- ✓ *Timer*: A timer may be set to execute a request such as: publishing in schedule, turning on/off device, and so on.
- ✓ *Tracking functions*: For searching the devices, topics, times,…Or showing the data of a selected deviced in a period.
- ✓ *Storing data*: The payloads can be kept in a database, which can be quite beneficial for collecting and analyzing data over a lengthy period of time.

- ✓ *User Management*: To manage the users's rights to access the resources on dashboard.

7.3. Making up the user interface:

Another thing need to be improved, that is the user interface. By making up the user interface, the dashboard can be more visual and user-friendly, such as: adding charts, icons, etc.

MQTT protocol is the primary protocol for all IoT applications, thus I hope that my work here will be recognized and utilized in future projects. In addtion, any improvements base on my project are welcome and this project is free for sharing and using.

References

[1] T. H. Team, "Introducing the MQTT Protocol - MQTT Essentials: Part 1," 12 January 2015. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/. [Accessed 15 May 2022].

[2] MQTT Community, "MQTT: The Standard for IoT Messaging," Updating. [Online]. Available: https://mqtt.org/. [Accessed 11 May 2022].

[3] HiveMQ, "Reliable Data Movement," HiveMQ GmbH, Updating. [Online]. Available: https://www.hivemq.com/. [Accessed 11 May 2022].

[4] I. Craggs, "MQTT Vs. HTTP for IoT," 16 May 2022. [Online]. Available: https://www.hivemq.com/blog/mqtt-vs-http-protocols-in-iot-iiot/. [Accessed 28 May 2022].

[5] M. Community, "MQTT Software," Updating. [Online]. Available: https://mqtt.org/software/. [Accessed 15 5 2022].

[6] "AWS IoT Core," Amazon Web Services, Updating. [Online]. Available: https://aws.amazon.com/iot-core/. [Accessed 11 May 2022].

[7] E. Foundation, "Eclipse Mosquitto - An open source MQTT broker," Cedalo, Updating. [Online]. Available: https://mosquitto.org/. [Accessed 11 May 2022].

[8] J. Yang, "kumquatt/mqttd," 3 December 2015. [Online]. Available: https://github.com/kumquatt/mqttd. [Accessed 15 May 2022].

[9] A. G, Actors: a model of concurrent computation in distributed systems, The MIT Press Classic, 1986.

[10] P. B. R. S. C. Hewitt, "A universal modular ACTOR formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73*, San Francisco, CA, USA, 1973.

[11] C. Hewitt, "Actor model of computation: scalable robust information systems," arXiv preprint arXiv:1008.1459, 2010.

[12] "Reactive Streams," 26 May 2022. [Online]. Available: https://www.reactive-streams.org/. [Accessed 30 May 2022].

[13] A. P. Team, "Introduction of AKKA Stream," 6 May 2020. [Online]. Available: https://doc.akka.io/docs/akka/current/stream/stream-introduction.html#motivation. [Accessed 30 May 2022].

[14] T. Tan Dung, "MQTT_Broker_AkkaModel," 30 June 2022. [Online]. Available: https://github.com/nicolas-le-petit/MQTT_Broker_AkkaModel/tree/Developing. [Accessed 30 June 2022].

[15] A. P. Team, "Classic FSM," 4 December 2020. [Online]. Available: https://doc.akka.io/docs/akka/current/fsm.html. [Accessed 25 May 2022].

[16] A. B. Ian Craggs, "Paho, Mosquitto and Security of MQTT," 12 March 2015. [Online]. Available: https://www.infoq.com/news/2015/03/paho-mosquitto/. [Accessed 9 September 2022].

[17] I. Cloudflare, "What is Transport Layer Security (TLS)?," Cloudflare, Inc., 2022. [Online]. Available: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/. [Accessed 09 September 2022].

[18] I. Cloudflare, "What is HTTPS?," 2022. [Online]. Available: https://www.cloudflare.com/learning/ssl/what-is-https/. [Accessed 09 September 2022].

[19] Lightbend, "Play 2.8.x documentation," Updating. [Online]. Available: https://www.playframework.com/documentation/2.8.x/Introduction. [Accessed 27 July 2022].

[20] P. Steve Burbeck, "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)," 4 March 1997. [Online]. Available: https://web.archive.org/web/20120729161926/http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html. [Accessed 28 July 2022].

[21] RJ45, "Simple Example of MVC (Model View Controller) Design Pattern for Abstraction," 8 April 2008. [Online]. Available: https://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design. [Accessed 28 July 2022].

[22] "What is AJAX?," Updating. [Online]. Available: https://www.w3schools.com/whatis/whatis_ajax.asp. [Accessed 30 September 2022].

[23] O. Organization, "MQTT Version 3.1.1 Plus Errata 01," 10 December 2015. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. [Accessed 11 May 2022].

[24] N. S. Gill, "Scalable and Responsive Applications with Akka | Quick Guide," 7 February 2022. [Online]. Available: https://www.xenonstack.com/insights/akka-framework-tools. [Accessed 20 May 2022].

[25] A. P. Team, "How the Actor Model Meets the Needs of Modern, Distributed Systems," 14 September 2020. [Online]. Available: https://doc.akka.io/docs/akka/current/typed/guide/actors-intro.html. [Accessed 25 May 2022].