VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Ilmari Eskelinen

SECURE MQTT PUSH NOTIFICATIONS IN A REACT ENVIRONMENT

Case DeviceVault

Business Information Technology
2022

VAASAN AMMATTIKORKEAKOULU
Tietojenkäsittely

TIIVISTELMÄ

Tekijä	Ilmari Eskelinen
Opinnäytetyön nimi	Turvalliset MQTT push-ilmoitukset React-ympäristössä
Vuosi	2022
Kieli	englanti
Sivumäärä	35
Ohjaaja	Päivi Rajala

Tämä opinnäytetyö käsittelee MQTT push-ilmoitusten toteuttamisprosessia ja hyötyjä. Opinnäytetyö myös käsittelee tämänhetkisiä markkinoilla olevia MQTT kojelautoja. Devatuksen DeviceVault-alusta toimii pohjana kaikelle kehitystyölle. Käyttöliittymä on toteutettu Reactilla ja TypeScriptillä. Palvelinpuolen viestien generointi tapahtuu Javalla. Työ sisältää myös työlle relevanttien teknologioiden esittelykappaleen.

Opinnäytetyön push-ilmoitusjärjestelmä toimii pohjana tulevalle push-ilmoitus-työlle DeviceVaultissa. Alustava pohja on turvallinen, laajennettava, luotettava ja käytettävä. Tämänhetkiset MQTT sovellukset ja alustat ovat riittäviä, muttei täytä standardia täysin automaatioidulle teollisuusjärjestelmälle. Markkinoilla on selvä rako jollekin otettavaksi.

Avainsanat MQTT, React, WebSocket, Java, push-ilmoitus

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Tietojenkäsittely
Business Information Technology

ABSTRACT

Author	Ilmari Eskelinen
Title	Secure MQTT push notifications in a React environment
Year	2022
Language	English
Pages	35
Name of Supervisor	Päivi Rajala

This thesis examined the implementation process of MQTT push notifications and their benefits. The thesis also investigated the different available MQTT dashboard options on the market currently. DeviceVault system by Devatus acted as the foundation on which everything was built. The user interface was made with React and TypeScript. The server-side message generation was done using Java. An introductory chapter about the relevant technologies is included in the Bachelor's thesis.

The push notification system laid out in this thesis is used as the foundation for future DeviceVault development. The foundation is secure, expandable, trustworthy, and usable. Current available MQTT apps and platforms on the market are sufficient, but not up to standard for a fully automated industrial system. A clear gap is present in the market for someone to take.

Keywords MQTT, React, WebSocket, Java, push notification

CONTENTS

TIIVISTELMÄ

ABSTRACT

1 INTRODUCTION 1

2 RELEVANT TECHNOLOGIES 2

3 DEVELOPMENT OF THE SYSTEM 8

3.1 Order of events within the system 8

3.2 Development of web app React application 9

3.3 Development of web app API 12

3.4 Development of hardware server, Posti and Common 14

3.5 Deployment of private MQTT broker 15

3.6 Trusting the page status 16

4 RESEARCH AND DEVELOPMENT OF DASHBOARD OPTIONS 19

4.1 Dashboard apps on the Google Play Store 19

4.2 Other MQTT dashboard platforms 21

4.3 Synopsis – thoughts on dashboard apps and platforms 24

5 RESULTS 25

REFERENCES 27

LIST OF FIGURES AND DIAGRAMS

Figure 1. Certificate-based authentication login in MQTT Explorer 6
Figure 2. Initialization of WebSocket connection in React 10
Figure 3. Reconnecting variable logic inside WebSocket hook 11
Figure 4. Logic for updating hardware list on content page 11
Figure 5. Preventing duplicate notifications with isActive 12
Figure 6. Examples of MQTT topic matches 13
Figure 7. Mosquitto MQTT broker access control list settings 16
Figure 8. Breaking circular reference by creating a new array of objects from a global list 23

Diagram 1. MQTT subscribe/publish model 5
Diagram 2. Order of events when hardware changes state 9
Diagram 3. Order of events when hardware server status changes 17
Diagram 4. Order of events when API loses connection to MQTT broker 18

Abbreviations and terms

Broker	An MQTT message transmitter. Stores unsent messages if retain is set to true
URL	Uniform Resource Locator. A pointer for unique resources on the internet. Resources can be HTML pages, CSS documents, images, files etc. An example of a valid URL is *www.google.com*
HTML	“HyperText Markup Language is the foundation of all web pages” (Codecademy 2022). HTML is used to organize text, images, videos, and all other content on the pages. HTML is the foundation on which everything on web pages is built.
IP address	An address unique to the device that identifies it on the internet or a local network. IP stands for "Internet Protocol," which is a collection of rules guiding the format of data sent via the internet or local network. (AO Kaspersky Lab 2022.)
Client	Client connected to any MQTT network on MQTT broker.
IoT	Internet of Things. Internet of applications or things.
I/O	Input/Output. Input and output of programmable logic. Used in MQTT and Node-RED.
Open-source	Source code that is freely available for use and modification. Products require permission for usage of the source code.
MQTT	Message Queue Telemetry Transport. An open-source publish/subscribe structure data transmission protocol.

DOM	Document Object Model. An interface that allows programs to interact with the page document or content by representing it as nodes and objects.
React.js	An open-source component-based JavaScript library used for state management and rendering stored state to DOM
Component	A function that accepts props and returns React elements describing what should be rendered on the DOM

1 INTRODUCTION

This thesis has three main objectives that were outlined by Devatus Oy. The first objective is to implement a hardware dashboard that can be used to show the detailed status of hardware. The second objective is to create a system dashboard summarizing hardware status for the entire system. The third objective is to update the current DeviceVault's various modules to generate, receive and handle MQTT messages. DeviceVault is a continuously accessible cloud platform designed for testing applications against multiple hardware and software variants without sending devices anywhere. All this can be done remotely with DeviceVault. (Devatus 2022.)

The responsibility for the first two objectives is to research different available technologies for displaying data from the MQTT messaging system and implementing two working dashboards. A hardware dashboard and a system dashboard are intertwined in this responsibility because they both receive messages from the same system and display the data in a dashboard app. The responsibility for the third objective is to update all modules responsible for generating, receiving, and handling MQTT messages. The work must be high-quality, reliable, and secure, in order to lay out a foundation for future development of push notifications in the system. Frontend React and TypeScript code needs to receive messages from the web app API module and then display the changes in real time. The module responsible for hardware needs to be updated to generate and publish messages into an MQTT broker. The module responsible for transmitting the information from other modules and user requests back to the user needs to be updated to subscribe to MQTT topics. This module also needs to be updated to include a WebSocket system, which enables the user to receive updates reliably and quickly instead of relying on an MQTT subscription client on the end user's machine. Separating the MQTT message subscription into web app API increases end users' performance and security of the system.

2 RELEVANT TECHNOLOGIES

This chapter is an introduction into the different key technologies and systems that are utilized in this thesis.

WebSocket

“The WebSocket API is an advanced technology that makes it possible to open a two-way interactive communication session between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply.” (Mozilla Corporation 2022)

Opening the WebSocket usually requires nothing from the user as it can and should be automated. During the opening process, the client and the server perform a handshake. The “Web” part in WebSocket is where the handshake happens. The objective of the handshake is to determine if both parties are suitable for two-way communication by negotiating terms of the connection. Either party can abort the handshake before its completion if the terms are unsuitable. The server must be cautious about the terms of the connection, otherwise cybersecurity issues can occur. (Mozilla Corporation 2022.)

React.js

React is a JavaScript library used for creating user interfaces. It was developed by Meta (formerly Facebook). React is open-source and is currently maintained by multiple parties including its parent and passionate developers and companies. According to a survey done by Statista, React is the most common web development library at 40.1 percent of developers using some version of React. According to the same survey React’s closest competitor Angular is only used by 23 percent of developers. (Statista 2022.)

React however is only concerned with DOM and state management and thus it requires support libraries like React Router to handle routing. Some other notable commonly used supporting libraries include React Redux and React Testing Library. React Redux is an open-source JavaScript library used for the management and centralization of application state. React Testing Library is a lightweight library that helps the developer create unit tests for the code, increasing maintainability and quality.

React is also often paired with a component library, which hastens development. Some notable component libraries are Chakra UI and Semantic UI. These libraries provide the backbone for basic features such as data display, forms and layout. The components featured in all component libraries take in props to perform various actions that would normally take development time. Selecting the correct component library is important as their features wary and their support might have ended. Semantic UI's support ended in 2018, but Chakra UI is still supported as of spring 2022. (Adebayo, S. 2022; GitHub Inc. 2022.)

Push notification

Push notifications are interactable pop-up messages that appear on users' browsers or devices independent of the device or browser. They serve as a quick communication channel for messages, offers or other updates between the source server and the end user. The technology is still relatively new as it was introduced in 2009 but is still underutilized compared to its massive potential for improving aspects like user experience and engagement. Push notifications can be utilized on four main platforms. The platforms are web push notifications, desktop push notifications, mobile app push notifications and push notifications on wearables.

Web push notifications are anything ranging from a hint to an offer. A hint notification can describe a change in status or a recommendation for the user to perform an action like for example, refresh the page to not miss updates. These type of push notifications are not opt-in. They commonly happen regardless of the user.

Desktop push notifications are limited to products that are installed onto the device. This can be a plugin like Honey notifying the user that there is a significant discount code for the store the user is currently browsing. These types of push notifications are opt-in only and have good engagement.

Mobile push notifications are also limited to user installed or pre-installed products like desktop push notifications. When the user opens the product, a unique identifier is registered for both the product and the device with the operating system's push notification service. The identifiers are shared with app publishers who can manually tailor and send push notifications with the goal to increase engagement. These types of push notifications are also opt-in only and have good engagement.

Wearable push notifications are also opt-in only as they are commonly synchronised from a source like a phone app the user has installed. The source of the notification can be anything from a calendar generating reminders to a brand app pushing advertisements. Luckily for the user, notifications can be disabled or silenced for opt-in options. Wearable push notifications have limitations like screen size and as a compromise they need to be short and concise. (Wingify 2022.)

MQTT

MQTT is a messaging protocol for Internet of Things (IoT). Its key features are its light weight and publish/subscribe messaging transport model that is good for remote devices. Implementation is designed to be quick and concise with minimal network bandwidth impact. It is used in many industries such as the automotive

industry, manufacturing industry, logistics, communications, oil, and gas. According to its creators, MQTT is even scalable to millions of devices. (mqtt.org 2022.)

Setup of an MQTT communication channel is simple and only requires two participants, a client, and a broker for messaging to start. The broker stores messages and handles subscriptions and publication onto the network. The client is subscribed to topics and publishes messages to topics. In diagram 1, a temperature sensor publishes temperature data onto the network. Two other clients are subscribed to temperature.

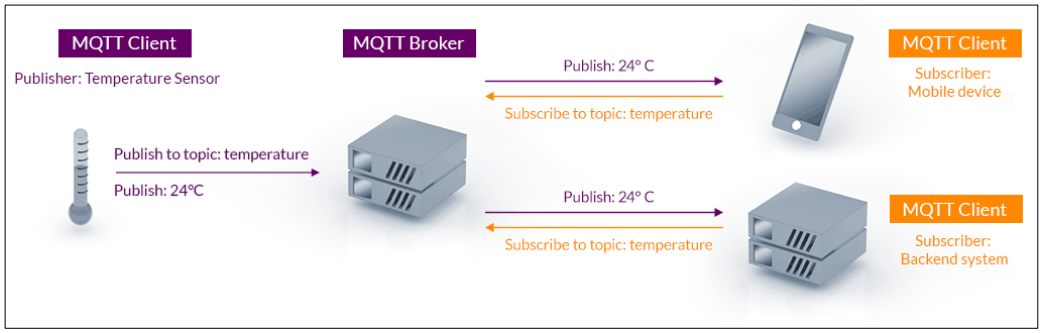

Diagram 1. MQTT subscribe/publish model

The broker can be configured to support encrypted TLS (Transport Layer Security) messaging which requires the client to perform some form of authentication. Some notable supported authentication options are OAuth, certificate-based authentication, and credential-based authentication. Configuring additional options for the broker process is done through a .conf file that the process reads on start-up. Certificate-based authentication requires multiple options to be configured. First *require_certificate* needs to be set to *true* for the broker to require certificates. Secondly three files need to be specified, first a certificate authority certificate, secondly the broker's own certificate and thirdly the broker's own key. Optionally *use_identity_as_username* and *allow_anonymous* can be set to *true* with certificate-based authentication to bypass requiring credential-based authentication. In figure 1, a client is configured to use the required files for a broker that has credential-based authentication enabled.

Figure 1. Certificate-based authentication login in MQTT Explorer

Credential-based authentication is easier to implement than certificate-based as it only requires two steps. In mosquitto, the first step is to add a new entry called password_file inside the config file which instructs the broker to load the file at the specified path. The second step is to hash the passwords. Hashing a password file is easy, as it can be done with a single command. For mosquitto, the command is mosquitto_passwd –U example.txt. The command converts plain text passwords into hashed passwords.

MQTT also has many features to support unstable connections. The first feature is called Quality of Service (QoS). QoS has three levels ranging from zero to two. QoS zero means at most once. QoS one means at least once. QoS two means exactly once. QoS level should be considered carefully, as it can affect the service. QoS zero should be used with stable connections where it does not matter if a message is lost. QoS zero is not compatible with message queueing or persistence either. QoS one should be used when the arrival of a message is critical to the experience and receiving multiple of the same does not affect the experience. QoS two should be used sparingly when it is critical for the application to receive the messages only a single time. QoS two is more performance intensive and slow than QoS zero

and QoS one, but it can be helpful if duplicate delivery is harmful to the application users. Queueing messages is available for QoS one and two. Queueing allows offline clients to receive messages after they become available again. However, queueing is only possible if the client has a persistent session. A persistent session is type of session, where the user has requested to maintain the subscriptions, messages, queued messages, and unacknowledged messages between connectivity. Many MQTT providers also support a technique called bridges. Bridges reroute the client's traffic in case of downtime on the current broker. Bridges are used to guarantee stability on the system if one of the brokers goes down. (HiveMQ GmbH 2022.)

3 DEVELOPMENT OF THE SYSTEM

This chapter goes into detail about the development of the modules included in the system. The environment consists of ten main modules, but the thesis' objectives are targeting the following five modules: Web app, Web app API, Posti, hardware server and common utilities. Web app is the react application displaying the system's collective information to the user. Web app API is the "postman" of the system receiving and delivering messages from hardware servers and Posti. It also processes commands and requests from web app users. Posti is a custom module that transmits files and information to other modules upon request. Posti got its name from its role in the system, but it is not affiliated with the Finnish postal service in any way. Common utilities are utilities shared across all different modules. All the modifications need to be high quality, secure, and expandable because the work is a foundation for future push notification development.

3.1 Order of events within the system

This chapter describes the events and actions within the system and their order. In diagram 2 a typical hardware connection status push notification chain consisting of the following steps is described:

1. Hardware changes state
2. Hardware server generates a message and publishes it to the MQTT broker
3. Web app API has a broad wildcard subscription active and receives message
4. Web app API converts message into a WebSocket message
5. Web app API evaluates which connections have an active subscription for the hardware
6. Message is sent to connections over a WebSocket connection

7. React page receives message over WebSocket connection. useEffect hook activates since one of the dependencies has changed. Code updates hardware's state.
8. Hardware changes state without refreshing or fetching

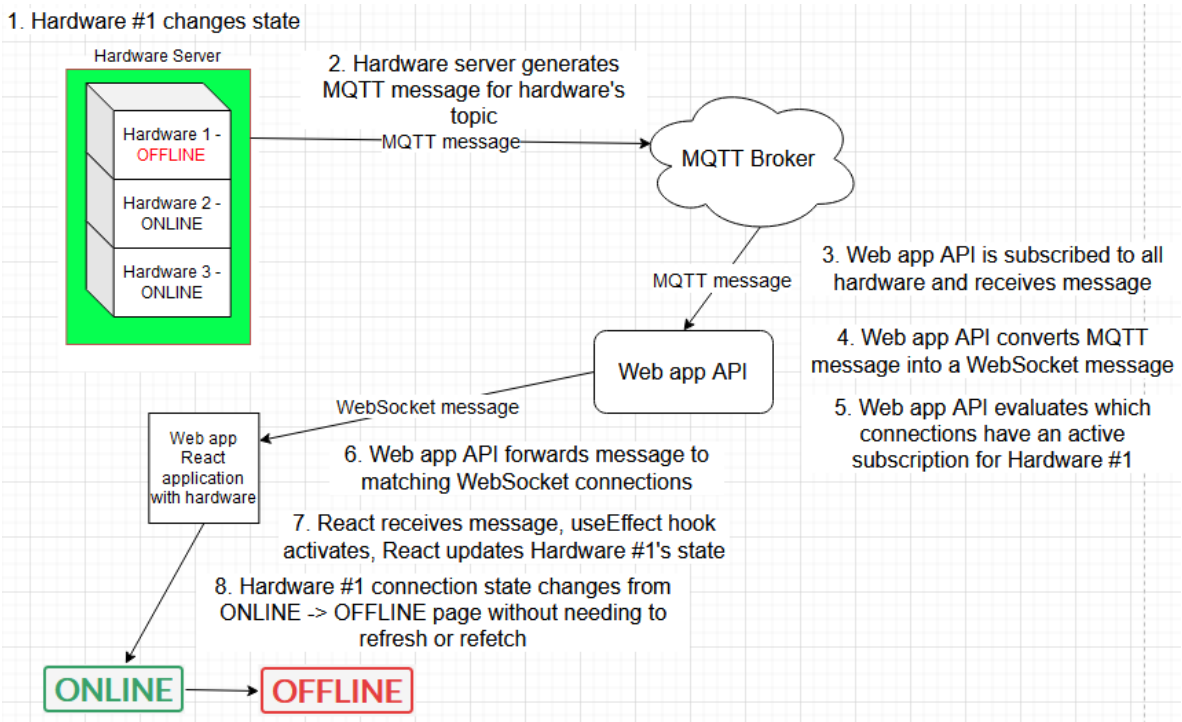

Diagram 2. Order of events when hardware changes state

3.2 Development of web app React application

The React application was updated to handle MQTT message payloads coming from Web app API through a WebSocket connection. The WebSocket connection is opened inside a custom hook called useWebSocket in figure 2. This custom hook uses React hooks to initialize, update and revive the WebSocket connection if needed. The React hooks in use are useEffect, useState and useRef.

```
16  export function useWebSocket() {
17    const webSocket = useRef<WebSocket | null>(null);
18    const [reconnecting, setReconnecting] = useState(false);
19    const [messageHistory, setMessageHistory] = useState<Array<WebSocketMessage>>([]);
20    const [latestWsMessage, setLatestWsMessage] = useState<WebSocketMessage>();
21    const intl = useIntl();
22    const showError = useShowError();
23
24    useEffect(() => {
25      webSocket.current = new WebSocket('ws://localhost:8200');
```

Figure 2. Initialization of WebSocket connection in React

UseRef is described in React.js's official documentation with the following text: "useRef returns a mutable ref object whose *current* property is initialized to the passed argument (*initialValue*). The returned object will persist for the full lifetime of the component." Using useRef means that the WebSocket connection is maintained between re-renders of the DOM, and it prevents the user's browser from opening multiple connections that slows both ends of the system. (Meta 2022.)

UseEffect is a hook that re-executes every time one of the dependencies changes in the dependency array. In figure 3 the variables *intl*, *reconnecting* and *showError* are used as dependencies. Out of the three dependencies listed only reconnecting is relevant to the component's logic. The other two dependencies do not change during runtime. Adding static dependencies is a good code practice to prevent weird runtime bugs from happening. It is also a requirement to not have missing dependencies in most code quality inspection tools.

UseState is used to initialize a Boolean called *reconnecting* to determine if the WebSocket is currently connecting to the WebSocket target server. If *reconnecting* is true, then we return out of the function on line 43 in figure 3.

```
41      webSocket.current.onclose = (e: CloseEvent) => {
42        if (webSocket.current) {
43          if (reconnecting) return;
44          showError(e, {
45            description: intl.formatMessage({ id: 'websocket_connection_close' }),
46          });
47          setReconnecting(true);
48          setTimeout(() => setReconnecting(false), 2000);
49        }
50      };
```

Figure 3. Reconnecting variable logic inside WebSocket hook

The hook is written so the latest message can be extracted and used in other components that subscribe to content. On the hardware page handling the push notifications is quite simple. In figure 4 the latest message is extracted on line 26 from the custom hook useWebSocket. The handling logic is again wrapped in a useEffect hook and *latestWsMessage* is used as a dependency. That means every time a new message is received, the hook will activate. In figure 4 on line 35 a new list of *hardwares* is being constructed by mapping through the old list and applying changes to it if the server identifier and serial number match on lines 37, 38 and 40. On line 44 the state of *hardwares* is being updated with the newly mapped list *newHardwares* using a useState hook called *setHardwares*.

```
25  export function HardwarePage() {
26    const { latestWsMessage } = useWebSocket();
27    const { loadHardware, isLoading, hardwares, setHardwares } = useLoadHardware();
28    const { instance } = useMsal();
29
30    useEffect(() => {
31      if (hardwares && latestWsMessage) {
32        const { payload } = latestWsMessage;
33        switch (latestWsMessage.subject) {
34          case 'connection': {
35            const newHardwares = hardwares.map((hardware) => {
36              if (
37                hardware.serialNumber === latestWsMessage.source.hardware &&
38                hardware.server.identifier === latestWsMessage.source.server
39              ) {
40                hardware.status.connection = payload.connection;
41              }
42              return hardware;
43            });
44            setHardwares(newHardwares);
45            break;
46          }
```

Figure 4. Logic for updating hardware list on content page

Web app was also updated to show if the status is out of date due to changes in the system. The reasoning for this is explained more in chapter 3.7. The first notification is an out-of-date notification on the page, which is a new custom hook that utilizes Chakra UI's toast component. The hook takes in options like description, which allows the component to be a generic reusable component for all warnings and not just for an individual use case or event. Inside useWebSocket the message subject is evaluated. If the subject is serverListChange, the out-of-date notification is shown to the user. The second notification is used for notifying the user about connection loss between Web app API and the broker. The underlying logic is the same as notifications for hardware server changes. The notifications are only ever shown once and do not multiply if the user has not cleared the previous one. In figure 5 preventing multiple notifications displaying at the same time is done using Chakra UI toast component's *isActive* function. IsActive takes in an integer, which it uses to compare if that toast is already active. Since this toast is only triggered through other hooks, the id is assigned as a static 1.

```
16      if (!toast.isActive(1)) {
17        toast({
18          id: 1,
19          title: intl.messages.useshowoutofdate_warning_title,
20          description,
21          duration: null,
22          position: 'bottom-left',
23          isClosable: true,
24        });
25      }
```

Figure 5. Preventing duplicate notifications with isActive

3.3 Development of web app API

Web app API module was upgraded to subscribe to MQTT topics on the Devatus MQTT broker. There are multiple reasons for implementing subscriptions inside web app API instead of each end user's React instance acting as their own MQTT client. The main reasons are security, performance, and authorization. The broker is configured to require private credentials for login, using system-wide credentials

in React would publicize the credentials. Performance is improved by having only the messages arrive to React that the user is viewing and eligible for. This also affects security in a massive way, as otherwise all users would receive network traffic for updates that they are not authorized for.

Web app API's subscription is done in a Java class called MqttMessageListener. The class opens a single connection to the MQTT broker. MqttMessageListener contains different subscriptions for different types of updates on the system. Multi-subscription structure allows Web app API to handle forwarding the updates to the users who are authorized to receive updates for that content. The hardware status topic the system subscribes to is dv/+/server/+/hardware/+/status/#. Figure 6 contains examples of how a typical hardware connection status change matches against different subscriptions. The /-symbols indicate topic levels. The +-symbols are single-level wildcards matching any symbols for the corresponding part. The first +-symbol is domain level, the second is server level and the third is hardware level. The #-symbol is a multi-level wildcard, meaning anything after /status/ will be a match. This means it can deliver any hardware's status changes located on any domain or any server.

Incoming MQTT message topic:	
dv/example_domain_1/server/example_server_1/hardware/example_hardware_1/status/connection	
Subscriptions:	
dv/+/server/+/hardware/+/status/connection	MATCH
dv/example_domain_1/server/example_server_1/hardware/example_hardware_1/#	MATCH
dv/example_domain_1/server/+/hardware/example_hardware_2/status/connection	NO MATCH
dv/example_domain_+/server/example_server_1/hardware/example_hardware_1/status/connection	NO MATCH
+/server/example_server_1/hardware/example_hardware_1/status/connection	NO MATCH

Figure 6. Examples of MQTT topic matches

Initially web app API was meant to be upgraded to work with the broker using certificate-based authentication, but it proved difficult and time consuming due to Java 8 not being fully compatible with the required keystore types and the MQTT client library in use not being fully compatible with the latest standards.

Instead of using certificate-based authentication Devatus felt credential-based authentication was good enough for the purpose of showcasing the system in a secure way by preventing unwanted users on the broker. The system will be updated to use certificate-based authentication after spring 2022.

3.4 Development of hardware server, Posti and Common

Hardware server

Hardware server module was upgraded to generate and publish MQTT messages when hardware status changes. The changes that trigger generation include changes in the hardware's connection or reservation. The system will be updated to have push notification support for hardware proxy status and action progress after spring 2022.

Posti

Posti module was upgraded to report on status of all mailboxes under all domains in the system. Mailboxes are destinations for Posti to deliver updates for and communicate with. Reporting is done by publishing information about the mailbox into the mailbox's corresponding topic. The information published is a JSON object, which contains name, type, status, and network of the mailbox. The messages are picked up by other clients to trigger events. Web app API uses these messages to notify end users about other mailboxes' changes.

Common

Common is a standalone module, which contains classes, definitions, utilities, and tools which all other modules implement and use. Common was upgraded to handle MQTT clients and publish messages. Common contains a Java class called MqttClientManager, which is used to manage MQTT clients for hardware servers and Posti. AbstractMqttMessage is a class which contains sub-objects, values, and

methods for a MQTT message. Whenever Posti or a hardware server needs to publish a message, they call a method inside AbstractMqttMessage called publishMessage. The method takes in a string topic and a byte array payload as parameters, which means it can be used globally by other modules if the data is formatted correctly.

3.5 Deployment of private MQTT broker

By the nature of the overall system and the data contained within the system there was an obvious requirement for an encrypted and private broker entity within the same system. The connection must be encrypted, and some form of authentication had to be implemented. In the end, credential-based authentication was selected for its ease of implementation and relative security. Certificate-based authentication was investigated but many different issues occurred in development due to the project being on Java 8. The issues are solvable, but I did not have time to fix them during my thesis and Devatus felt credential-based authentication was good enough for the purposes of demoing the system. The system will be updated to use certificate-based authentication after spring 2022.

The broker is configured using a *mosquitto.conf* file that the program reads upon startup. The broker is configured to have two roles. The role configuration is done using two text files. The first text file is a password file, which contains pairs of usernames and passwords. The second text file is an ACL (Access Control List), which controls access to actions for roles. Dashboard role is meant for dashboard apps and the role is read-only. Devicevault role is used for all the proprietary modules that are a part of the system. In figure 7 the final configuration is seen.

```
# These affect clients with username "dashboard" or "devicevault".
# Usernames are configured in password_file
user dashboard
topic read #

user devicevault
topic readwrite #
```

Figure 7. Mosquitto MQTT broker access control list settings

3.6 Trusting the page status

This chapter explores problems regarding the trust the user can have in the web app's status. DeviceVault system is complex, and many parts can fail. This presented problems in the trust the user can have in the Web app's status. Web app can become unsynchronized for multiple reasons. A common reason for an unsynchronized page is new hardware servers coming online or going offline. Other reasons can be internet outage or connection loss to either Posti or the configured broker.

Hardware server changes

To solve hardware server trust issues, a new quality-of-life (QoL) feature was added. The feature is a generic out-of-date notification that is triggered in the web app if an incoming message's subject is *serverListChange.* In diagram 3 Posti publishes a message whenever a mailbox's status changes, and web app API is subscribed to pick them up. Web app API forwards a modified message to all web app WebSocket connections. The modified message's subject is *serverListChange*. In web app's useWebSocket hook the code evaluates if the message's subject is *serverListChange.* If the result is true, the out-of-date notification is shown.

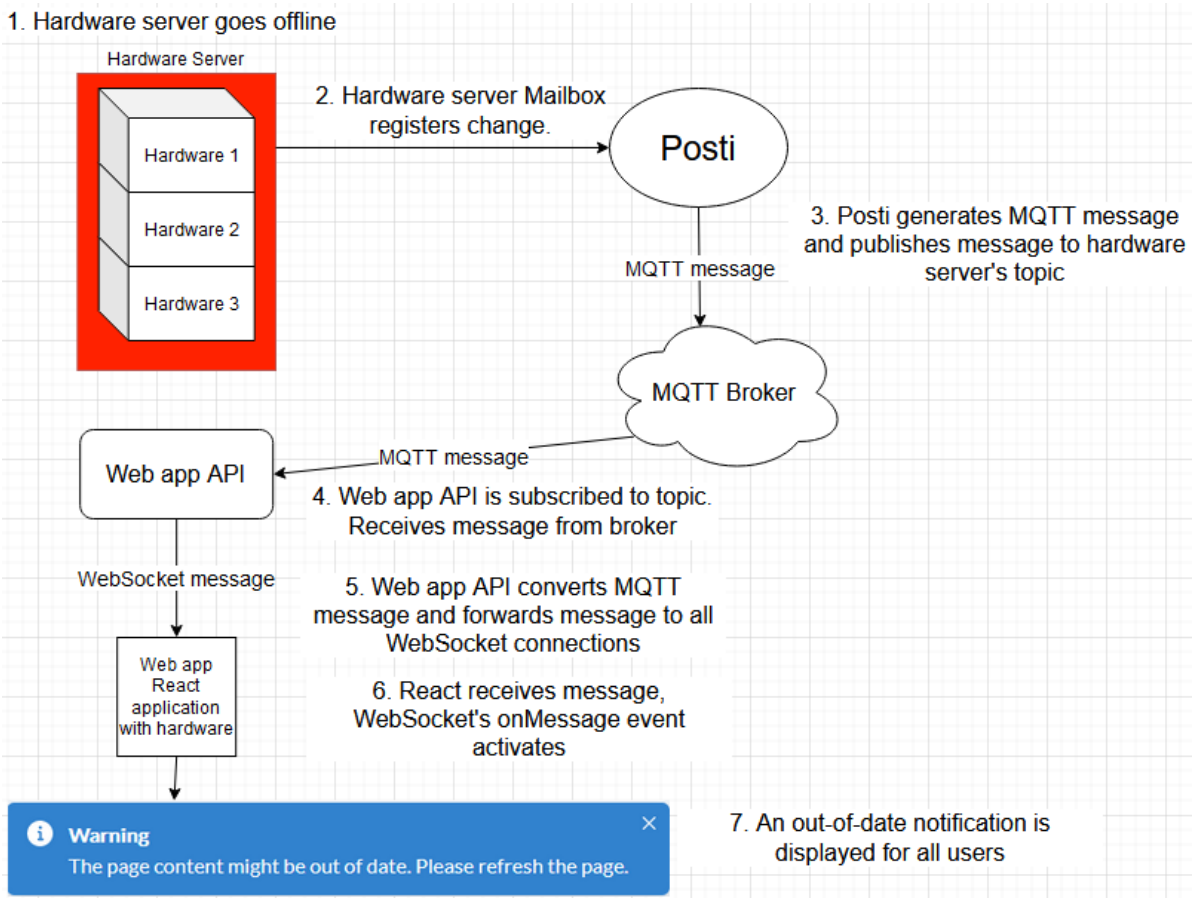

Diagram 3. Order of events when hardware server status changes

Currently there is a minor problem within this event stack; users that are not subscribed to the specific Mailbox still receive the messages and the out-of-date notification is displayed. The system will be updated to solve this after spring 2022.

Connection loss to broker

The code library used in web app API to communicate with brokers as a client has support for certain events. One of these events is connection lost. With the connection lost event, connection loss to the broker can be detected and a message can be sent to the web app React application. After the message arrives to web app, a notification is rendered informing the user of connectivity issues in the system. In diagram 4, the order of events when web app API loses connection to MQTT Broker is described.

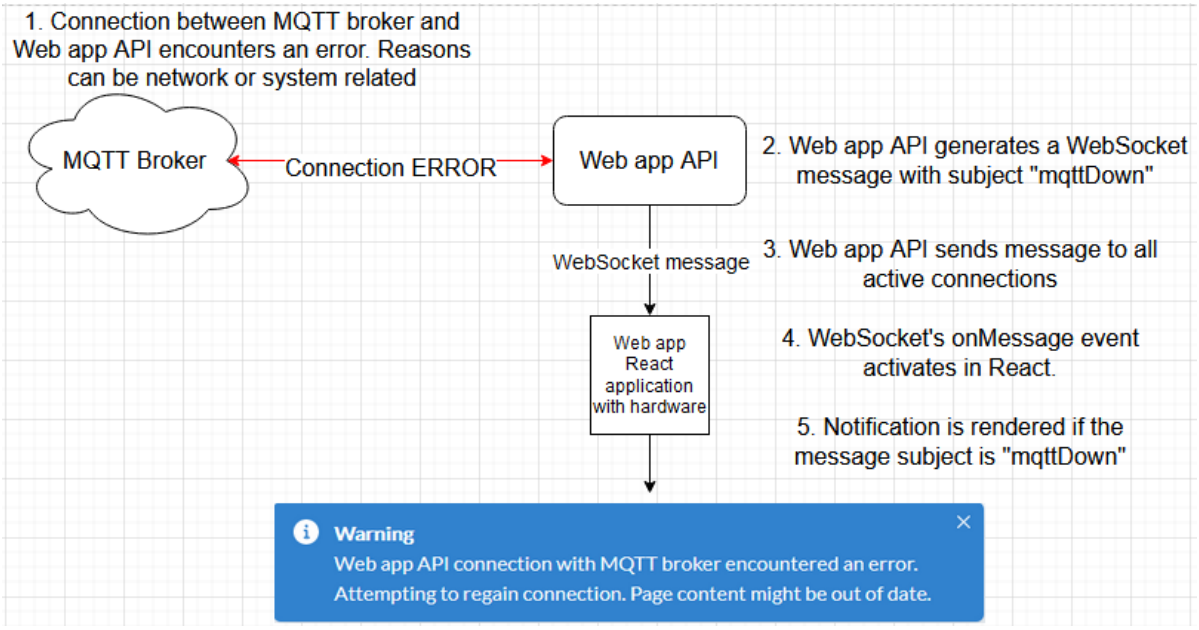

Diagram 4. Order of events when API loses connection to MQTT broker

4 RESEARCH AND DEVELOPMENT OF DASHBOARD OPTIONS

This chapter investigates the available options that are currently on the market and what they lack to fulfil the objectives outlined in the requirements. The chapter includes in-depth findings and analysis at the end. The options presented are all public and free to use. All of them are easy to configure and they do work for simple use cases like home light bulb control or room temperature reporting. Simple IoT projects commonly have predefined definitions and simple I/O logic. Three out of seven options did not support wildcards in any way and only 2 supported them in ways that support automation.

4.1 Dashboard apps on the Google Play Store

This chapter investigates five dashboard apps found on Google Play Store. The apps are MQTT Dash, IoT MQTT Panel, MQTT Dashboard, MQTT Dashboard – IoT and Node-RED Controller and Linear MQTT Dashboard

MQTT Dash (IoT, Smart Home)

MQTT Dash is a free app developed by Routix Software. It has over one hundred thousand downloads and a 4.8 rating. The app can be found here: https://play.google.com/store/apps/details?id=net.routix.mqttdash. It supports saving connection details for multiple brokers. The primary reason this app failed is the lack of support for displaying complex data and no support for wildcard subscription. The range of available tiles is disappointing too, as the app only supports text, switch, range, multi-choice, image, and color.

IoT MQTT Panel

IoT MQTT Panel is a free app developed by Rahul Kundu. It has over one hundred thousand downloads and a 4.4 rating. The app can be found here: https://play.google.com/store/apps/details?id=snr.lab.iotmqttpanel.prod. The app supports saving multiple broker connections into memory for easy use. The

app also had the best support for diverse types of panels to use for data display. It has fifteen distinct types of panels ranging from user-controlled panels to graphs and charts. This app only supported wildcards in a single panel called text log. Text log displays messages from a specified wildcard subscription.

MQTT Dashboard

MQTT Dashboard is a free app developed by Lapetov. It has over five thousand downloads and a 4.2 rating. The app can be found here: https://play.google.com/store/apps/details?id=com.lapetov.mqtt. The app supports saving connections to different brokers in storage for quickly changing between brokers. This app has average level support for different data display options. It did not, however, meet the requirements set by Devatus for a single reason. The reason for failure is the app does not support wildcards. If a wildcard subscription is entered into a topic field, the app crashes and becomes unresponsive. Future attempts to launch the app also result in a crash. The only way to restore the app to a functional state is to locally clear all stored data. This app is the poorest option available due to its instability and average tile support.

MQTT Dashboard – IoT and Node-RED controller

MQTT Dashboard – IoT and Node-RED controller app is a free app developed by Vetru. It has a 4.3 rating and over fifty thousand downloads. The app can be found here: https://play.google.com/store/apps/details?id=com.app.vetru.mqttdashboard. The app's free version does not support saving connection details for multiple brokers. However, the app can be upgraded to a Pro version for 3.99 €, which unlocks this feature. The app has good documentation on its website about the possibilities, visuals, and logic of the tiles available. The tile selection overall is better than average, but still lacking. There is no multi-line graph support. A single-line chart is supported but is too burdensome to use since it has no memory. It

requires an array of integers to draw the line every time the line needs to be updated. The array history could be implemented with something like Node-RED using state and function blocks.

Linear MQTT Dashboard

Linear MQTT Dashboard is developed by ravendmaster. It has a 4.2 rating and over 10 thousand downloads. The app can be found here: https://play.google.com/store/apps/details?id=com.ravendmaster.linearmqtt-dashboard. The app does not support wildcard subscriptions. The app has better than average support for different types of tiles. It does not support saving connection details to storage. The app has many good features like the graph tile which supports four real-time graphs and time axis adjustment. Another good feature is programmable onReceive and onShow events. Another good feature is the ability to export a configuration file for reuse. The events are programmed using JavaScript. The worst feature is how the graph tile produces a lot of noise and spam, which drains the device's battery and uses excessive amounts of device resources.

4.2 Other MQTT dashboard platforms

This sub-chapter investigates Flespi and Node-RED as MQTT dashboard platforms.

Flespi

Flespi is a free MQTT platform owned by Gurtam (Flespi 2022). Flespi's home page is https://flespi.com/. Flespi has a good range of high-end complex tiles and gadgets to use. The documentation found online is good enough to start developing dashboards. Flespi also supports wildcards in subscriptions. Flespi requires users to use a login token to get access into their private MQTT network. Flespi is not compatible for DeviceVault due to it requiring the use of Flespi's own private broker and having no support for user-created code like Node-RED.

Node-RED

Node-RED is a flow-based programming interface used to build logic for MQTT message traffic using visual programming. Node-RED is developed by OpenJS Foundation and its contributors. Its home page is https://nodered.org. Node-RED provides a flow editor for browsers that makes it easy to wire together flows using a wide range of nodes in the palette. Flows can be deployed in a single click. (Node-RED 2022.)

Node-RED is the most complex out of the options investigated. On top of having a flow editor, it features a real time rendered user interface and support for custom JavaScript code. It requires a host machine to use. The host machine hosts a local network session that can be accessed by other devices on the same network. The flow editor and final user interface can both be accessed by other devices in the network with a browser by going into their respective URLs. The URLs start with the host machine's IP address. With default configuration settings the flow editor's URL ends in /flow and the user interface in /api/ui. Node-RED supports custom template HTML output from Node-RED and although by default it is not visually pleasant, it can be improved with external libraries, imports, and CSS. It also has support for basic features like text fields, gauges, and charts.

Node-RED has two weaknesses, the first is the lack of easy-to-use global storage of previous messages. The second is the functions and blocks inside the flow can only access the newest incoming *msg* variable. To get around these weaknesses, unique server and hardware entries are stored into global lists. The lists are called *serverList* and *hardwareList.* The function that handles storing is similar to the function on web app's hardware page handling updates to state. After every message, a new list of hardware is built from those lists. However, serverList or hardwareList cannot be assigned to the message object directly because it would create a circular reference. To work around the circular reference problem, in figure 8 an array of objects is generated, and the list is assigned to *msg.hardwareList* or

msg.serverList. After the circular reference problem is fixed a template HTML block uses the list inside *msg* variable to create a table with all the hardware and server entries and their status.

```
const hardwareList = global.get("hardwareList")

const newHardwareList = hardwareList.map((hardware) => {
    let newObj = {
        "serialNumber": hardware.serialNumber,
        "server": hardware.server,
        "payload": hardware.payload,
        "subject": hardware.subject,
    }
    return newObj
})

msg.hardwareList = newHardwareList;
return msg
```

Figure 8. Breaking circular reference by creating a new array of objects from a global list

At this point, the data is complete and can be converted into other forms. Converting is useful to make the data simpler and usable with the apps in section 4.1. Earlier in development this is how dashboards were envisioned to integrate with DeviceVault data. A new section was added to the flow diagram which assigned global *hardwareList's* length property to *msg.payload*. Node-RED's MQTT out node only publishes data inside msg object's payload property.

Node-RED did meet Devatus' expectations for a tool to convert messages into values that can be monitored from dashboard due to its support for converting messages into monitorable values using the built-in JavaScript function blocks. However, it is not used in production as a simpler solution was developed. The replacement is an integrated publishing method inside Posti that updates the number of mailboxes under a domain every time Posti updates an individual mailbox. This solution was much simpler and cut out Node-RED as an extra dependency in an already complex system.

4.3 Synopsis – thoughts on dashboard apps and platforms

The apps and platforms that are currently available on the market do not meet the standard required for a fully automated system that can store the status of previous messages and update a list of hardware as updates happen. However, all of them work and are clearly usable for simpler work or home IoT projects. The quality of apps found on Google Play Store are mixed but some of them are quite good and suitable for most projects. If I had to choose an option based on the findings, I would choose Node-RED. Node-RED is the best because it supports JavaScript function blocks which allow the user to alter the data and build custom HTML mixed with Angular. The template HTML output from Node-RED by default is not visually pleasant, but it can be improved with external tools and CSS. By default, it also has sufficient support for basic features like text, gauges, and charts for simpler use cases. Node-RED also supports wildcards but is lacking in the automation aspect. Automation is doable with Node-RED, but it requires a hacky unofficial solution. There are libraries and imports that patch up Node-RED's weaknesses.

It is clear there is a gap in the market for someone to take. IoT and MQTT are incredibly useful technologies with almost unlimited potential. The current commercial tools are not suitable for large-scale dashboards with a dynamically updated user interface. Wildcard subscriptions were the largest piece missing from most apps. The current market is missing an app or platform that can create good looking understandable dashboards with wildcard subscriptions and automation in mind. Complex systems with many objects constantly changing require automation and wildcards to work and remain maintainable.

5 RESULTS

All the objectives and requirements outlined by Devatus were fulfilled. Push notifications were successfully integrated into the already existing DeviceVault framework. Push notifications improve the system's usability and trustworthiness. Trustworthiness is improved with multiple different notifications that are triggered by events happening in the system. Usability is significantly improved by keeping the status of the system up-to-date live, without needing to constantly refresh or poll the server for updates. The foundation is high-quality, secure, and expandable. The foundation will be expanded after spring 2022 to support push notifications for other more complex parts of DeviceVault like proxies and hardware actions.

The broker is private thanks to credential-based authentication and an access control list. Certificate-based authentication can later be added to improve security; however, it may prove time-consuming due to problems with Java 8. Access control list restricts dashboard role from publishing into the network, which prevents noise.

The work is going through final review, polishing, and testing in spring 2022. After the review process it will be integrated into the system as an integral part, and it will be eventually rolled out to the customers in a release.

The research and testing on the MQTT dashboard options proves that there is a gap in the market. The current dashboard options available on the market lack automation and ease of use for large-scale operations. The results of this thesis are useful for anyone interested in the relevant technologies, developers implementing similar systems, and entrepreneurs looking for a gap in the market. The work is useful for documenting complex systems, understanding the use cases of push notifications, and for understanding the current market for MQTT dashboards.

The most difficult aspect of the work was maintaining scope and designing a secure and expandable system. The easiest part was documentation due to the great effort spent on designing and brainstorming. Overall, the thesis project has been very successful from an objective standpoint for the future of DeviceVault and for mapping the state of the MQTT dashboard market.

REFERENCES

Adebayo, S. 2022. Create accessible React apps with speed. Accessed 5.4.2022. https://chakra-ui.com/

AO Kaspersky Lab. 2022. What is an IP address – Definition and Explanation. Accessed 23.3.2022. https://www.kaspersky.com/resource-center/definitions/what-is-an-ip-address

Codecademy. 2022. Learn HTML. Accessed 23.3.2022. https://www.codecademy.com/learn/learn-html

Devatus. 2022. Device Vault. Accessed 6.4.2022. https://www.devicevault.io/

Flespi. 2022. About us. Accessed 29.3.2022. https://flespi.com/about-us

GitHub Inc. 2022. Semantic UI. User Interface is the language of the web. Accessed 5.4.2022. https://semantic-ui.com/

HiveMQ GmbH. 2022. Quality of Service 0,1 & 2 – MQTT Essentials: Part 6. Accessed 23.3.2022. https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/

Meta. 2022. React. Hooks API Reference. Accessed 23.3.2022. https://reactjs.org/docs/hooks-reference.html

Mozilla Corporation. 2022. The WebSocket API (WebSockets). Accessed 29.3.2022. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

mqtt.org. 2022. MQTT: The standard for IoT Messaging. Accessed 22.3.2022. https://mqtt.org/

Node-RED. 2022. Low-code programming for event-driven applications. Accessed 29.3.2022. https://nodered.org

Statista. 2021. Most used web frameworks among developers worldwide, as of 2021. Accessed 22.3.2022. https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/

Wingify. 2022. What are push notifications?
Accessed 22.3.2022. https://vwo.com/push-notifications/