

Damir Mustafin

Building Analytics Plugin for Erlang MQTT Broker

Bachelor's Thesis
Information Technology

2017



**Kaakkois-Suomen
ammattikorkeakoulu**

Author (authors)	Degree	Time
Damir Mustafin	Bachelor of Engineering	May 2017
Thesis Title		
Building Analytics Plugin for Erlang MQTT Broker		39 pages 10 pages of appendices
Commissioned by		
Sampo Software OY		
Supervisor		
Reijo Vuohelainen		
Abstract		
<p>The purpose of this thesis was to research and develop a piece of software that would allow to collect analytics data from a proxy server that is built for Internet of Things. The main work was done around EMQ broker, which was selected as a proxy server for this project. EMQ serves requests over MQTT protocol, an alternative to traditional HTTP. It is written in Erlang programming language that belongs to a family of functional programming languages that gives high fault-tolerance and concurrency when serving requests on the server-side. The following proxy was selected to work with APIInf, an API management platform, for a number of reasons that are described in the thesis.</p>		
<p>The methods used for researching and implementing the system involved learning an Erlang programming language in order to develop a plugin for the broker. In addition, EMQ provided a pre-defined template that was used as a starting point for the development process. The gathered analytics data was then visualized via the web user interface that involved creating a simple analytics dashboard that shows requests distribution over time. In order to navigate over the big amounts of data, filtering options were added to the dashboard.</p>		
<p>The outcome of the development was a number of functioning components that were performing the requested task. As the initial goal of this project was to fully integrate EMQ with APIInf, the work in that direction will continue and this thesis describes steps that are left for the integration.</p>		
Keywords		
Erlang, MQTT, JavaScript, NodeJS, EMQ, IoT, REST, API, Elasticsearch		

CONTENTS

1	INTRODUCTION	4
2	ERLANG PROGRAMMING LANGUAGE	4
2.1	HISTORY OF ERLANG	5
3	HTTP	9
4	MQTT PROTOCOL	10
4.1	PUBLISH/SUBSCRIBE PATTERN.....	11
4.1.1	Scalability.....	13
4.1.2	Message filtering.....	13
4.2	CONNECTION ESTABLISHMENT	13
4.2.1	Open a connection.....	14
4.2.2	Publish, subscribe and unsubscribe.....	16
4.3	MQTT TOPICS.....	20
4.3.1	Topic structure	20
4.4	QUALITY OF SERVICE.....	21
5	PRACTICAL PART	23
5.1	SOFTWARE USED.....	23
5.1.1	APIInf	24
5.1.2	Erlang MQTT Broker.....	27
5.1.3	EMQ analytics plugin	28
5.1.4	Elasticsearch installation and setup	31
5.2	ANALYTICS DASHBOARD	32
6	RESULTS	35
7	CONCLUSIONS AND FUTURE WORK	38
	REFERENCES	39
	APPENDICES.....	40

1 INTRODUCTION

Nowadays, a lot of small digital services and startups like web-sites, mobile applications, servers and databases by the matter of fact, unexpectedly, can be highly loaded. That high usage can be represented as unpredictable flow of users or a DDOS attack. For that reason, it is quite important to setup the environment and security at least at a minimum level in the beginning, support and improve it while the user base grows.

Setting up environment can mean a lot of things. But the main focus of this thesis is studying Erlang MQTT Broker, a tool which by its nature is built to be massively scalable, distributed, extensible broker for Internet of Things that is able to handle millions of concurrent clients and extending it with a plugin for analytics data collection. The gathered data should be then visualized via the dashboard.

The requirement for this project came from a company called SampoSoftware. APIInf is the project the team has been working on. It is an API Management platform, fully open-sourced and built around API Umbrella – a proxy that does all the heavy lifting for managing APIs. In order to extend the functionality towards multi-proxy support and to enhance product offering, it was decided to add EMQ as a second proxy for API Management.

This thesis describes the essential language and protocols that were used not only for analytics plugin development, but also for the tools on top of which the plugin itself was built on. The practical part introduces to the development process, overall system configuration and describes software that was used more specifically.

2 ERLANG PROGRAMMING LANGUAGE

According to Wikipedia (2017a) Erlang is a functional programming language that was designed for writing concurrent programs that “run forever”. Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing, they are also lightweight and belong to the language, not the operating system. Erlang also allows

changing code and adding patches “on the fly”, so that there is no need to restart the program since it can be updated during runtime. These mechanisms simplify the construction of software for implementing non-stop systems. (Joe Armstrong. A History of Erlang, Ericsson AB)

Since Erlang belongs to the family of functional programming languages, it has a number of essential differences in routine development approaches and methods. All variables in Erlang are “single-assign”, meaning that once the variable is assigned to certain value, it cannot be re-assigned to different value. The closest example from the other programming languages that have similar feature, would be “final” keyword in Java. Due to lack of variable re-assigning, loop construction in the language makes no sense, as there is no way to change a variable over successive iterations. All control that could imitate looping mechanisms in the language is done by recursive calls. In addition to Erlang’s syntax and internal language features, it has a huge standard library which includes a HTTP and SSH client, server, SNMP capabilities and embedded NoSQL database server.

Erlang was selected as a programming language for a plugin development, mostly because EMQ Broker itself is written on Erlang (and supports only plugins written on the same language), and the advantages of the language listed above only strengthened this choice. In fact, EMQ broker was selected as a suitable proxy for APIInf because it is written in Erlang and takes advantage of all its features like high-availability and fault-tolerance.

2.1 History of Erlang

The actual Erlang development took place in 1986 at the Ericsson Computer Science Laboratory. The team was given a task of investigating programming languages suitable for programming the next generation of telecom products – they needed to provide a better way of programming telephony applications. At that time telephony applications were very different from the ones that we have today. They used to have a lot of problems that ordinary programming languages were designed to solve. These applications by their nature are highly concurrent, a single

switch must handle tens or hundreds of thousands transactions. Transactions are distributed and the software is expected to be highly fault-tolerant.

Joe Armstrong (A History of Erlang, Ericsson AB) states: *“When the software that controls telephone fails, newspapers write about it, something which does not happen when a typical desktop application fails.”*

It is very important that in telephony applications, software is updated while the system runs without loss of service that is occurring. It must also operate in the “soft real-time” domain and with strict timing requirements for some operations, but with more simplified timing for other classes of operations.

Another requirement for Erlang was virtually zero down-time. The rise of popularity of the Internet at that time and the need for non-interrupted availability of services has extended the class of problems that Erlang can solve. For example, building a non-stop web server, with dynamic code upgrade, handling millions of requests per day is very similar to building software to control a software exchange.

Usually the software for call control is modeled using limited number of state machines that carry state transitions in response to protocol messages. From the software point of view, the system behaves as a very large collection of parallel processes. At any point in time most of the processes are waiting for an event caused by the reception of a message or the triggering of a timer. When the event occurs, the process does some amount of computation, changes state, sends messages to other processes and then waits for the next event. The amount of time spent on computation is very small. Any switching system, on the other hand, should be able to handle hundreds of thousands of very lightweight processes where each of them performs little computation. Also, software errors in any process should not be able to crash the system or damage other processes in the system. The issue with this was that in a system having too large number of processes there was a need to protect processes from memory corruption problems. In languages with pointers, processes are protected from each other using memory management hardware, allocating lower limit of the memory to the process. It also means that the memory

requirements for an individual process can be very small and that all memory for all processes can be in the same address space without needing memory protection hardware. In Erlang, processes are actually a part of the language itself, not the underlying operating system. It simply does not need most of the services of the operating system, since the language provides both – memory management and protection between parallel processes. Other operating system services, such as resource allocation and device drivers needed to access the hardware are written in C and dynamically attached to Erlang runtime.

Three properties of a programming language were central to the efficient operation of a concurrent language or operating system:

1. The time to create a process
2. The time to perform a context switch between two different processes
3. The time to copy a message between two processes

Joe Armstrong (A History of Erlang, Ericsson AB) states: *“The earliest motivation for Erlang was ‘To make something like PLEX, to run on ordinary hardware, only better.’”*

Ideas for Erlang were heavily inherited from PLEX and AXE design. PLEX is a special-purpose, event-driven, real-time programming language dedicated for AXE telephone exchanges. One of such ideas was the possibility to change code “on the fly”. Thus, AXE had “pointer” problems like parallel manipulation of a large number of telephone calls. Memory requirements for each call were variable and memory was allocated using linked lists and pointer manipulation. This led to a big number of errors. This was the inspiration of the process and garbage-collected memory strategy used in Erlang.

The final key idea that comes from the AXE/PLEX culture is that the failure of a process or hardware should only affect the transaction that is being processed and that all other operations should remain up and running. Because of this, the Erlang design was simplified and hanging pointers between processes were removed.

Message passing was implemented by copying message buffers between the memory spaces of the different processes. To achieve fault-tolerance, the team denied all the ideas of sharing resources between processes because of the difficulties with error handling and decided that all processes should have enough local information to keep running if something fails in another part of the system. So, in order to make system reliable, the data was copied between processes so that the processes had enough data to continue running by themselves, if other processes crash.

Due to AXE/PLEX history and its internal architecture, a list of requirements has been provided for each new programming language to have. (Joe Armstrong. A History of Erlang, Ericsson AB.)

This list included the following:

- handling a very large number of concurrent activities
- actions to be performed at a certain point of time within a certain time
- systems distributed over several computers
- interaction with hardware
- very large software systems
- complex functionality such as feature integration
- software maintenance without stopping the system (reconfiguration, etc.)
- stringent quality and reliability requirements
- fault tolerance both to hardware failures and software errors

These requirements were pretty straight forward, and existing systems solved some of them at that time already, sometimes in operating system and sometimes in application libraries. Even though, in order to meet all the requirements, it was decided to continue experiments in prototyping telecom applications with all the available programming languages at that time.

3 HTTP

According to Wikipedia (2017b), HTTP (Hyper Text Transfer Protocol) is a request-response based protocol in the client-server computing model. HTTP operates on an application layer on the Internet protocol suite and defines how messages are formatted and transmitted and what actions should the web-server take in response to various commands. The web-browser is a well-known example of a client, while an application running on a computer hosting a web-site could be a server.

HTTP is often called a Stateless Protocol due to its nature of executing each request independently, without any knowledge of the requests that came before it. HTTP provides a wide variety of status codes and error messages that are attached to each response that came from the server as a payload. These status codes define the precise scenario that was either successful or unsuccessful while requesting a web-page or some document. User agent (UA), a web-browser for example, should deliver and display any related entity to the user. Official HTTP status codes are divided into five classes and each of these classes indicate as follows:

- **1xx** status codes are **informational responses** that indicate that the request was received and understood.
- **2xx** status codes are **successful responses** that indicate that the action requested by the client was received, understood, accepted and processed successfully.
- **3xx** status codes class is a **redirection**. It indicates that the client must take an additional action to complete the request.
- **4xx** status codes are **client errors**. These status codes are intended for situations in which the client seems to have erred.
- **5xx** status code are **server errors** that indicate cases in which the server is aware that it has encountered an error or is otherwise incapable of performing the request.

Some most common status codes are: **200** that is standard “OK” response for successful HTTP requests, **403** refers to “Forbidden” and tells that the user might have a lack of permissions for a resource, **404** or “Not Found” means that the requested resource could not be found and **500**, “Internal Server Error”, is a generic error message given when an unexpected condition was encountered and no more specific message is suitable.

In addition to official status codes, there is a number of unofficial ones that are not specified by any standard and usually are extensions provided by software or services like web-servers or reverse-proxies.

4 MQTT PROTOCOL

According to Wikipedia (2017c), MQTT is a light weight publish/subscribe based messaging transport protocol (application layer protocol) designed for Machine to Machine and Internet of Things (M2M/IoT) communication. MQTT is very lightweight and binary protocol, which excels when transferring data over the wire in comparison to protocols like HTTP, because it has only a minimal packet overhead. Also, HTTP is often too verbose, sends a lot of meta-data, and TLS handshake is required every time the connection is requested. Another advantage of MQTT is that it is really easy to set it up on a client side. This is useful for constrained devices with limited resources (mobile phones, smart watches).

MQTT was created by Andy Stanford-Clark (IBM) and Arlen Nipper in 1999. They created a list with specifications which the future protocol should have:

- simple to Implement
- provide a Quality of Service data delivery
- lightweight and bandwidth efficient
- data agnostic
- continuous session awareness

IBM used it for some time internally until they open sourced the specification in 2010. And since then MQTT became widely used by a lot of different companies.

In 2013 a committee was built to standardize MQTT. And, at the end of 2014 MQTT became a standard.

MQTT client is a client that has MQTT library running on it and is connected to the MQTT broker over the network. This could be any kind of device, starting from a micro controller up to full-fledged server that has a TCP/IP stack and speaks to MQTT over it. MQTT client libraries are written for a huge variety of languages and platforms (like Android, C, C++, C#, Go, iOS, Java, JavaScript, NodeJs, Erlang) and are available freely. The full list of MQTT libraries is also available on GitHub.

MQTT broker is a counterpart to MQTT Client and is a core of the MQTT publish/subscribe mechanism. The MQTT broker is primarily responsible for processing, receiving and filtering messages. The broker also decides who is interested in particular messages by the topic and distributes them accordingly to all the subscribed clients. It also handles session information of all the connected clients, subscriptions and missed messages that are delivered due to the absence of the client's connectivity. In addition, the broker is also responsible for authenticating and authorizing clients. Depending on the specific library, the broker is also extendable. This feature allows integrating custom layers of authentication, authorization and integration into backend systems. Integration plays a very important role as the broker component itself exposed to public access and handles a lot of clients passing messages. In general, the broker operates as a hub for all the connected clients which must be highly scalable, integratable into backend systems, easy to monitor and most importantly failure-resistant.

4.1 Publish/Subscribe pattern

The biggest difference between MQTT and some other request-response protocols is that MQTT uses the publish/subscribe pattern. It is an alternative to the traditional client-server model where the client communicates directly with an endpoint. The publish/subscribe method operates in a way that it decouples a client sending a message (publisher) from another client (or more clients) receiving the message (subscriber). This way a publisher and a subscriber do not know about the existence of each other. However, there is a one more component (broker) who both

the subscriber and publisher know about. The broker filters all incoming messages and broadcasts them accordingly.

Let's take a look at the example of publish/subscribe operation in the Figure 1.

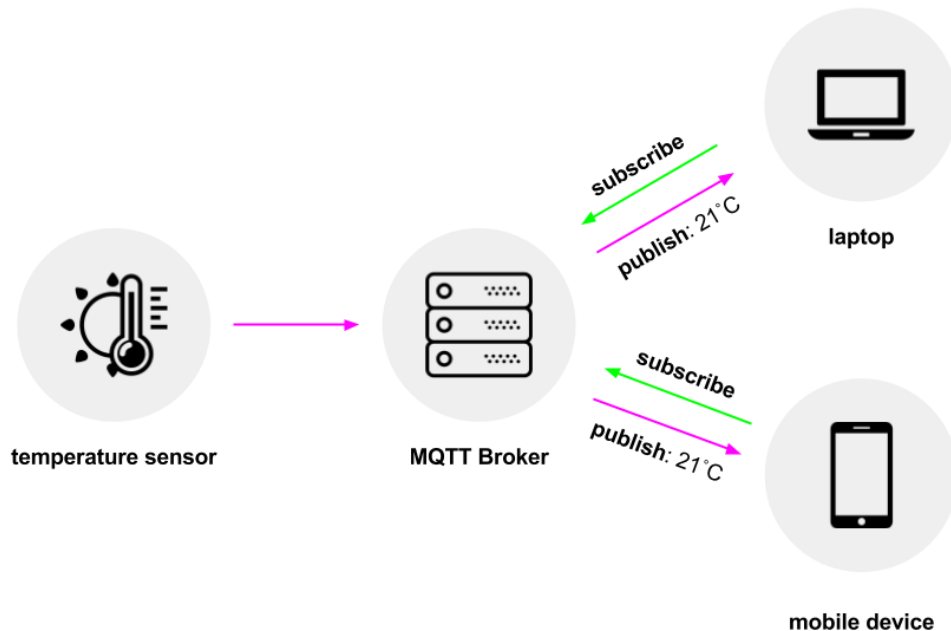


Figure 1. MQTT Publish/Subscribe pattern diagram

Referring to the diagram, there is a temperature sensor that publishes temperature value regularly to devices that might be interested in this data. In addition, there is a MQTT broker in the middle that manages all the connections. There are two parties here: MQTT clients (temperature sensor, laptop and mobile divide) and a broker. All of these clients have only one MQTT connection to the broker and can publish and subscribe. It is more or less similar to magazine subscription, for example. Once subscribed, the new magazine will be delivered once the new release is published. Mobile device subscribes to a temperature value and gets it from the sensor directly. The client can be a subscriber and publisher at the same time. However, as mentioned earlier, this approach decouples the publisher and subscriber which can be differentiated in more dimensions which are represented in the list:

- space decoupling: the publisher and subscriber do not know each other (by IP address and port for example)
- time decoupling: Publisher and subscriber do not need to run at the same time
- synchronization decoupling: Operations on both components are not suspended during publishing or receiving

It is possible that only certain clients receive certain messages during message filtering. The decoupling has three dimensions: space, time, and synchronization.

4.1.1 Scalability

The publish/subscribe approach provides a great scalability because operations on the broker running in parallel and processes are event-driven. Message caching and intelligent routing of messages are important ways for improving the scalability.

4.1.2 Message filtering

Message filtering is needed for the broker to filter all the incoming messages so that each subscriber gets only messages it is interested in.

- Subject-based filtering: based on a subject (or topic) which is a part of each frame
- Content-based filtering: based on a specific content filter-language
- Type-based filtering: filtering based on the type/class of the message (event)

4.2 Connection establishment

MQTT is built on top of TCP/IP of the OSI layer model and requires both the client and the broker that are using it to support TCP/IP stack. Each client has a connection to a broker and this allows to push connections to other clients and they get messages in real time (in addition to network latency). Each client can open only one connection and is capable of pushing messages.

4.2.1 Open a connection

The connection is initiated by the MQTT client sending a CONNECT message to a broker. The broker will respond with a CONNACK and a status code after which clients are officially connected. Once the connection is established, the broker will keep it open until the client sends a disconnect message or loses the connections.

Table 1. MQTT CONNECT packet

Contains	Optional	Example
clientId	no	client01
cleanSession	no	true
Username	yes	"bob"
Password	yes	"Passw0rd12345"
lastWillTopic	yes	"/bob/will"
lastWillQos	yes	2
lastWillMessage	yes	"unexpected exit"
lastWillRetain	yes	false
keepAlive	no	60

The table above represents the example CONNECT message that is sent from the client to the broker to initiate a connection. If the message contains errors (some parameters are missing or incorrect) or it takes too long from opening a network socket to sending it, the broker will close the connection. This good security feature helps to avoid malicious clients that can slow down the broker. Referring to the message structure, there are three required parameters that the message must contain. They include the following:

- **clientId** in a client identifier used to identify each of the clients connected to the MQTT broker. The ClientID should be unique for each broker.
- **cleanSession** is a flag that indicated whether the client wants to establish a persistent connection or not.
- **keepAlive** defines the time interval in milliseconds to which the client is committed to by sending regular PING requests to the broker. The broker responds to PING with PING response and this allows both parties to determine that the other side is still alive and reachable.

In addition to the required parameters, it is possible to pass more parameters, which are optional by default, to extend the configuration of the connection. Here are some of them:

- **Username** and **Password** are credentials used to authenticate and authorize a client. It is important to mention that a password is sent by plain text by default. Although it is possible to encrypt or hash it by implementation or to use TLS underneath.
- **Will** message is a part of the last will and testament feature of MQTT. It allows notifying other clients, when the client disconnects without sending DISCONNECT message beforehand.

Table 2. MQTT CONNACK packet

Contains	Example
sessionPresent	True
returnCode	0

Once the broker receives a CONNECT message, it is obligated to respond with a CONNACK message. The CONNACK message only contains two fields:

- **sessionPresent** is a flag that represents whether the broker already has a persistent session of the client from previous interactions. It helps the client to determine if it has to subscribe to topics, or if these are still stored in his session.
- **returnCode** is the connection acknowledgement flag. It tells the client, if the connection attempts were successful or, if not, what is the issue.

Here is a table to all possible status codes that the broker can respond with:

Table 3. MQTT CONNACK packet

Return Code	Status	Reason
0	Connection Accepted	-
1	Connection Reused	Unacceptable protocol version
2	Connection Refused	Identifier rejected

3	Connection Refused	Server unavailable
4	Connection Refused	Bad username or password
5	Connection Refused	Not authorized

4.2.2 Publish, subscribe and unsubscribe

After the client device has successfully connected to the MQTT broker, it can start publishing messages. As message filtering in MQTT is based on topics, each message must contain a topic. When a broker receives the message, it then forwards it to interested clients. The message publishing can be described in three steps:

1. client sends a publish message to a broker
2. the broker reads the publish message
3. the broker processes messages by determining which clients have subscribed to this topic and sending the message to those who is interested (subscribed)

Table 4. MQTT PUBLISH packet

Contains	Example
packetId	1234
topicName	"topic/1"
qos	1
retainFlag	false
payload	"temperature:28.5"
dupFlag	false

- **Topic Name** is a string value that defines a topic to publish message to, structured hieratically, using forward slashes as "topic level" separators.
- **QoS** is a number that represents a Quality of Service level which defines the guarantee of the message delivery to the end, other client or broker.
- **Retain Flag** determines, if the message will be saved by the broker for the specified topic as a last known useful value. New clients subscribed to that topic will receive the last retained message on that topic instantly after subscribing.

- **Payload** is the actual content of the message. MQTT supports sending data of any kind, such as images, texts, encoding, encrypted and binary data.
- **Packet Id** is a unique identifier used to identify a message in a message flow.
- **DUP flag** indicates that the message is a duplicate and is sent again due to other end not sending a message of acknowledgment.

In order to subscribe to a topic and receive relevant messages, a client needs to send a SUBSCRIBE message to the MQTT broker. Subscribe message contains the packet identifier and list of subscriptions, as shown in the Table 5:

Table 5. MQTT SUBSCRIBE packet

Contains	Example
packetId	1234
qos1	1
topic1	“topic /1”
qos2	0
topic2	“topic /1”
...	...

- **Packet Id** is a packet identifier that is used to identify a message in a message flow.
- **List of Subscriptions** represents a list of the QoS & topic pair which are later used by the broker to route messages to clients. SUBSCRIBE message can contain any number of QoS & topic pairs. The topic value can contain wildcards that are useful for subscribing to different topic patterns, which gives more flexibility. If subscriptions are overlapping, the broker selects a message for delivering based on the highest QoS value.

After the client sends a SUBSCRIBE message to a broker, it will be confirmed and sent back to the client as an acknowledgement in the form of the SUBACK message.

Table 6. MQTT SUBACK packet

Field	Example
packetId	1234

returnCode 1	0
returnCode 2	2

- **Packet Id** is a packet identifier
- **Return Code** a code returned for each of the QoS/topic pair broker received from the SUBSCRIBE message. It represents a code to acknowledge each topic with the QoS level granted by the broker. If the subscription was not success-ful because of lack of permissions for example, the broker will respond with a failure return code.

Table 6. Possible return codes

Return Code	Return Code Response
0	Success - Maximum QoS 0
1	Success - Maximum QoS 1
2	Success - Maximum QoS 2
128	Failure

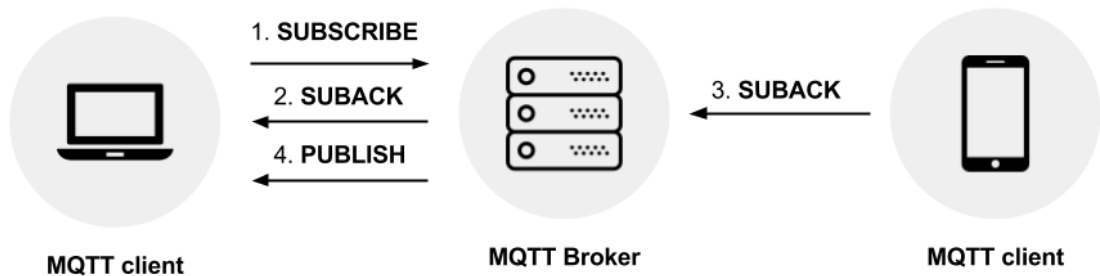


Figure 2. MQTT subscribe pattern

The UNSUBSCRIBE message is used to delete existing subscriptions of a client on a broker. The message structure is similar to the SUBSCRIBE message, it also contains a packet identifier and list of topics from which unsubscribe to.

Table 7. MQTT UNSUBSCRIBE packet

Field	Example
packetId	1234
topic1	"topic/1"

topic2	"topic/2"
...	...

- **PacketId** is used as identifier to detect the acknowledgement of an UNSUBSCRIBE message which will contain the same ID.
- **List of Topics** represents a list of topics, the client wants to unsubscribe from. The QoS value is not provided in the UNSUBSCRIBE message, since the topic will be unsubscribed regardless of the QoS level it was initially subscribed with.

The UNSUBACK message contains only the packet identifier is sent after broker has received and processed the request to unsubscribe. When the client receives the UNSUBACK from the broker, it can assume that all the subscriptions in the UNSUBSCRIBE message are deleted.

Table 8. MQTT UNSUBACK packet

Field	Example
packetId	1234

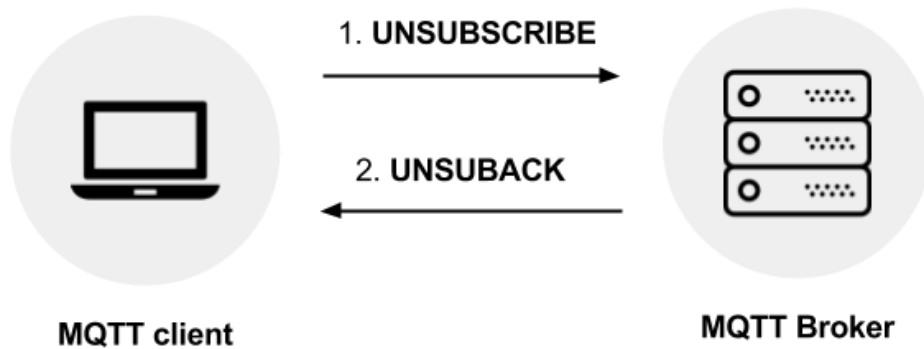


Figure 3. MQTT UNSUBSCRIBE pattern

4.3 MQTT topics

Beyond the publish/subscribe pattern that is one of the core features of MQTT, there is a concept called topics. The entire message routing in a broker is built on topics. Topics have hierarchical structure which gives flexibility and ease of use.

The best explanation to topics would be an example that is popular in IoT these days. Let's assume that there is a house with a number of rooms and floors in it. Each room in this house has a sensor which measures temperature in the room regularly and sends it to the broker. For a living room, which is located on a ground floor, the topic would look like the following

myhome/groundfloor/livingroom/temperature

The topic values are constructed like URLs and contain different topic levels. This is a core concept of MQTT, and it is very important because routing of messages is only built on topics. This way the broker knows to which topic to send the messages and the client knows to which topic to subscribe to.

4.3.1 Topic structure

The topic's structure looks very much like a URL. The topic level separator is slash ("*/*"), and we have different topic levels that can be arbitrary topic values. When it happens so that the client subscribed to too many topics, the topic will look messy and big. For this reason, in order to minimize the topic, the single-level wildcard can be used. Single-level wildcard represents only one topic level. In case of the example with the house and temperature values, the route may look like the following:

myhome/groundfloor/+/temperature

The client subscribes to topic that looks like this (when single-level wildcard is used), the client will receive messages for all the topics that are located on this level. The "+" character can be used more than once in a route.

Another possibility if the client wants to subscribe to all hierarchies below a specific topic label is a multi-level wildcard. It can only be used in the end of the route, as follows:

myhome/groundfloor/#

This makes topics a flexible concept and every client that subscribes to them, can use wildcards. When the client is publishing, no wildcards are allowed. Wildcards can be used only on the subscribe level.

4.4 Quality of Service

QoS defines a level at which one of the parties (a client or broker) guaranteed to deliver a message to another party. It is a major feature in the MQTT protocol, since it helps to avoid bottlenecks in unreliable networks as well as empowers a client to choose a QoS level depending on the network reliability and application logic. There are three QoS levels in MQTT:

- QoS 0 – at most once
- QoS 1 – at least once
- QoS 2 – exactly once

QoS 0 is a minimal level. It sends a message only once and the message will not be acknowledged by the receiver.



Figure 4. MQTT QoS 0 publish pattern

QoS level 1 guarantees that the message will be delivered to the other party at least once. But it is possible that the message will be delivered more than once. The sending party will keep sending the message over and over until it receives a PUBACK message. The PUBACK message contains only the packet id and is used to compare its value with the packet identifier in each packet for determining, if the message was received or not.

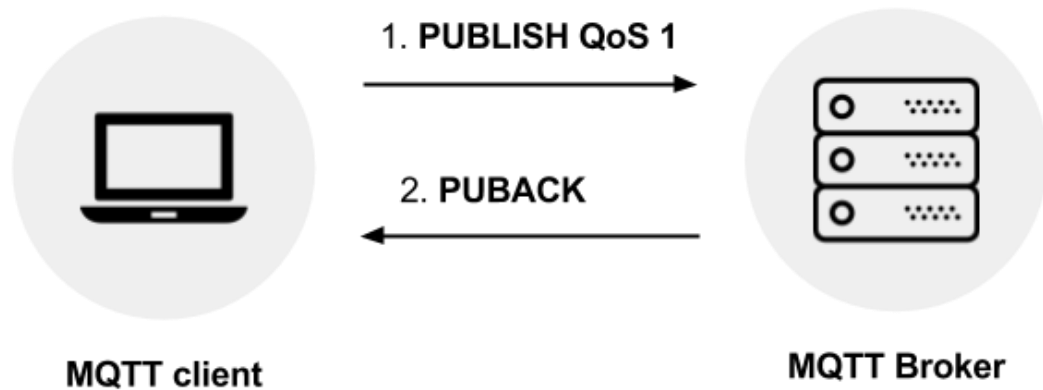


Figure 5. MQTT QoS 1 publish pattern

QoS 2 is the highest QoS level and it is the safest and slowest one. It guarantees that the second party of the communication will receive a message only once.

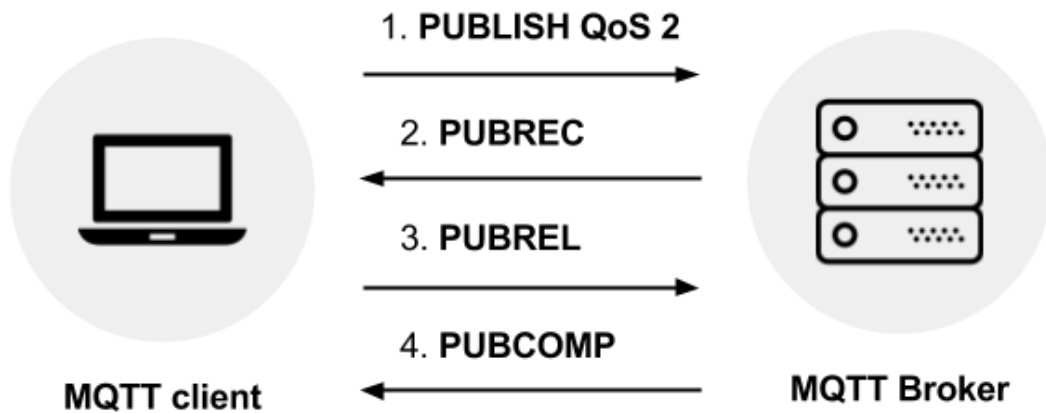


Figure 6. MQTT QoS 2 publish pattern

Once the PUBLISH message with QoS 2 is received by one of the parties, it will process it and respond with acknowledgement in the form of the PUBREC message. It also will also keep the reference to the packet id, so it is possible to later recognize duplicated messages. When the sender receives PUBREC it discards the initial PUBLISH because it knows that the other party has successfully received it. The receiver then will store PUBREC and respond with PUBREL. When PUBREL is received by the second party, it discards every stored state and responds with PUBCOMP. After the sender gets the last message and the flow is completed, both sides can be sure that the message has been delivered and the sender knows about it. If it happens so that some message is lost and was not received by one of the parties, the sender is responsible for sending the same message once again after a reasonable amount of time.

5 PRACTICAL PART

The goal of the practical part was to develop a plugin for EMQ that must collect analytics data from EMQ and store it somewhere. The decision of the data storage should be used, was made in favor of elasticsearch. The reason for choosing this tool came from the amount of features, flexibility, scalability, ease of use and speed that EMQ provides, such as comprehensive query language that elasticsearch supports and allows developers to fetch not only stored data but also aggregated buckets that are useful for analytics dashboard.

5.1 Software used

The software that has been used during the practical part is listed below and includes software versions which were used:

- Ubuntu Server 16.04.1 LTS 64-bit
- Oracle JDK
- ElasticSearch v5.2.0
- EMQTT v2.0.7
- EMQ-PLUGIN-APINF v2.1.1 (based on EMQ-PLUGIN-TEMPLATE v2.0.7)
- Meteor Boilerplate

5.1.1 APIInf

APIINF is an API Management platform that has been in the development over the last two years as a Sampo Software product. APIINF is an open source tool, that allows API administrators to host their APIs and gives a possibility to provide dynamic documentation powered by Swagger, comprehensive API usage analytics and more. Initially, APIInf was built around API Umbrella. Then, its architecture has been redesigned from the bottom up, and the core feature that was added there is a multi-proxy support. This means that administrators that hosts their APIs via APIInf are able to add new, and switch between proxies. This gives more flexibility and more ease for testing and further development.

There are three core APIInf features:

- API Catalog (As shown on the Figure 7)
- Analytics Dashboard (Comprehensive dashboard with many filtering options)
- API Documentation

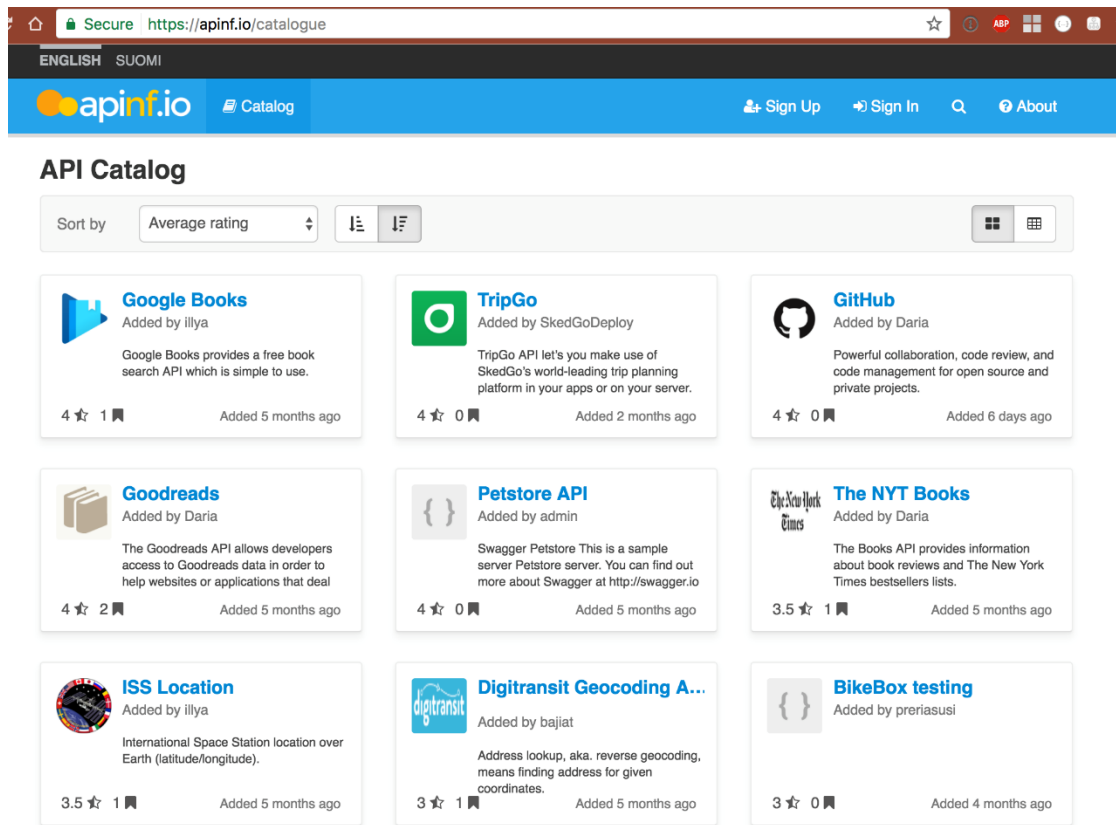


Figure 7. APIInf API Catalog

The practical part was implemented around APIInf dashboard in order. The dashboard represents API usage over time. Charting libraries that handle visual data rendering It are powered by D3 – JavaScript front end library for building Data Driven Documents. To give more clear understanding of the dashboard design and what it contains I will divide it into four parts, as follows:

- Filtering
- statistics
- charts
- data table

These components are shown in the Figures 8 and 9.

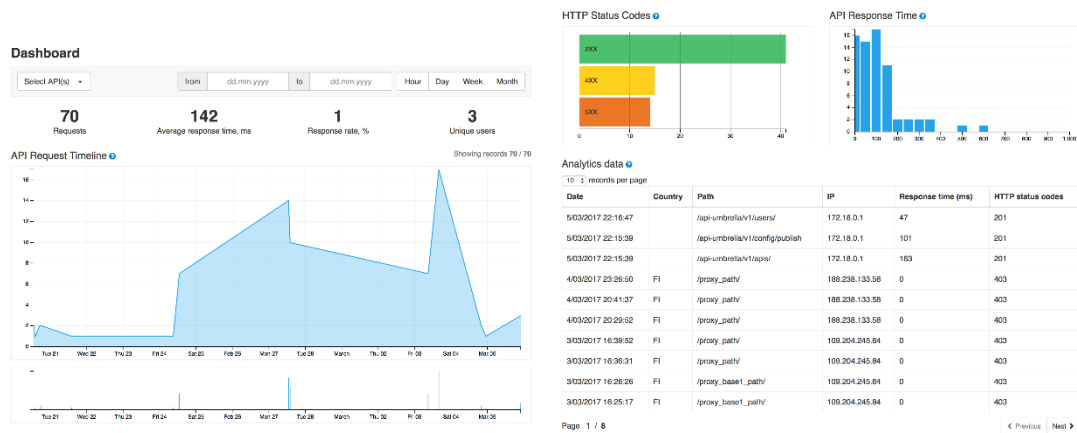


Figure 8-9. APIInf Analytics Dashboard

The filtering component is the one of the controllers that manipulate the data to be shown on the dashboard. There are multiple options for filtering. First of all, filtering by specific API(s). Selecting one or more APIs will instantly trigger a filter function that will re-render charts depending on a query. It is also possible to filter data within a specific time range by selecting the date to filter by using from and to date-pickers. Finally, there is an option to select custom granularity for data – hour, day, week or month. Higher granularity like week or month values are useful on a large scale if it is needed to structure the records over a wide range like one year. Selecting “hour” in this case would give very precise overview which might not be needed, while “month” can give much more general understanding of API usages.

The second component is nothing but an average summary of different data dimensions: total number of records in a memory, average response time, response rate and number of unique users.

The third component contains four charts. There are four of them – range and overview charts that show requests over time, a bar chart that groups HTTP status codes and visually compares them to each other, and lastly a row chart that shows grouped statistics for response time distribution in milliseconds. The overview chart is used for manual range selection on the chart using an in-built brush tool.

In addition, there is a data table that lists currently selected data providing much more detailed information about each request such as the exact time stamp when

the request took place, county from where the request came from, URL path, IP address, response time and HTTP status code.

The dashboard also uses a crossfilter – a JavaScript library that gives real-time data reactivity over all the components that handle filtered data. That means that if either data is filtered by a specific chart or via filtering component, all charts, data-table and statistics summary will at once receive the actual filtered data without page refresh.

APIInf dashboard gets its analytics data from the proxy it is connected to. Data is stored in the elasticsearch data store. In case of the API Umbrella proxy, the elasticsearch instance is built inside it and represented as an adapter.

5.1.2 Erlang MQTT Broker

EMQ is an Erlang MQTT Broker which is selected as a second proxy in addition to API Umbrella. EMQ is built as modular software and by default has a list of plugins that can be used and enabled during run time using either terminal (command line interface) or user interface.

According to the EMQ documentation, the EMQ broker internal design philosophy is represented in the following five points:

1. Focus on millions on concurrent MQTT connections and routing MQTT messages between clustered nodes.
2. Embrace Erlang/OPT, soft real-time, low-latency, concurrent, and fault tolerant platform.
3. Layered design: Connection, Session, PubSub and Router layers.
4. Separate the Message Flow Plane and the Control/Management Plane.
5. Stream MQTT messages to various backends including MQ or databases

5.1.3 EMQ analytics plugin

EMQ provides a template plugin which already has preconfigured setup and allows easily extending EMQ workflow. As the plugin was implemented within the APInf project, the template plugin was forked and stored as an organization repository on GitHub. Initially the plugin template has a list of pre-defined hooks that are called when client does a certain action like connect, disconnect, subscribe and unsubscribe. The broker uses a chain-of-responsibility pattern to implement a hook mechanism. This is a mechanism that gives more than one object an opportunity to link receiving objects together. So the hook functions registered to a hook will be executed one by one.

Table 9. EMQ hooks available in plugin template

Hook	When triggered
on_client_connected	client connects to the broker successfully
on_client_disconnected	client disconnects from the broker
on_client_subscribe	before the client subscribes to topics
on_client_unsubscribe	client unsubscribes from topics
on_session_created	session is established
on_session_subscribed	after the client(sessions) subscribed to a topic
on_session_unsubscribed	after the client(sessions) unsubscribed to a topic
on_session_terminated	the session is terminated
on_message_publish	the MQTT message is published
on_message_delivered	the MQTT message is delivered
on_message_acked	the MQTT message is received

Each of these hooks, as shown in Table 9, needed to be extended to send a bucket of data to the elasticsearch. There is a Erlang library called ESIO that provides an HTTP client API for communication with elasticsearch. The library is open-sourced and distributed under the MIT license.

Each application or library written on Erlang that follows community development principles and conventions is most likely to contain a dependency manager called

“rebar”. This is done by adding a rebar.config file to the project root directory, even if the application does not have any dependencies. In addition to dependencies list, the configuration file can also contain general package information like an application name, description, version and environment variables. ESIO documentation has a clear installation instruction, API and usage examples.

Once the dependency is added to the project and compiled, `esio` instance is globally available over the project. In order to collect the information that we might be interested in, a simple function was written that should make following steps: open the connections, check if the connection did not return any errors, write to the elasticsearch and close the connection.

```

42
43 write_to_es(Log) ->
44     esio:start(),
45     {ok, Sock} = esio:socket("http://127.0.0.1:9200/"),
46     Id = uuid:to_string(uuid:uuid1()),
47     esio:put(Sock, "urn:es:mqt:analytics:" ++ Id, Log),
48     esio:close(Sock).
49

```

Figure 10. Code example: Erlang function that writes data to elasticsearch

Figure 10 shows the implementation of the function described above. In Erlang, functions are defined in the following manner: first comes the function name followed by brackets with can take a number of parameters separated by a comma. In our case it is only one parameter which is a tuple that contains the data to be saved. The function body is separated from the previous part with an arrow “->”. Function calls, variable assignments, etc. are written inside function body and separated by comma. In order for a compiler to separate the function from other function or code, it must be ended with a fullstop. The `esio:start` function initializes esio instance. Next, on line 45 the connection is opened to a socket and due to Erlang's nature for dynamic pattern matching it is possible to perform an error check right on the same line. The next line is needed for generating a unique identifier, so that it is possible to differentiate each record later if needed. `esio:put` is used to write documents to elasticsearch mapping of the index “mqt”, identifiable

by key which is contained by “Id” variable. This function has three parameters: the actual socket connection, that we opened earlier, query string that defines the index to where to write the record, and lastly, the data tuple. The last line closes the connection.

Next, it is needed to call that function when certain event occurs. Each hook, when called, contains data needed for logging. In fact, each hook contains data that depends on a context of each hook, so it is different for each of them. Also, there were some fields that are not needed for analytics, because this kind of data does not really make any sense and is not valuable for a user or an administrator viewing this analytics (ClientId for example). For that reason, it was decided to develop a schema, that will not be actually used anywhere, but will help to group all the available fields and put them into one record, because all records should have similar schemas as close as possible. Table 10 demonstrates data fields that have been used in the schema.

Table 10. Single document schema

Field	Type	Availability	Description
date	string	yes	Timestamp when the record was saved
type	string	yes	Title oh hook that has been called
username	string	yes	Username of authenticated client
topic_table	object	depends on hook	List of topics to subscribe to
topic_and_opts	object	depends on hook	Topic and options
reason	string	depends on hook	Reason for session terminate
message	object	deoends on hook	Message send via Publish

It is worth noting that the date field represents a timestamp when the record has been saved. It is not available within the hook and was generated manually simply using the `erlang:localtime()` function.

5.1.4 Elasticsearch installation and setup

Elasticsearch is an open-source platform for distributed search and analysis of data in real time. The reason for choosing this platform came from number of features that it provides such as its flexibility, scalability, ease of use and speed. Elasticsearch supports RESTful operations, allowing to manipulate data using HTTP methods (GET, POST, PUT, DELETE, etc.) in conjunction with an HTTP URI (/collection/entry). In addition to that, elasticsearch also provides a complex, JSON-style domain-specific query language (Query DSL) which makes it easy to fetch not only stored data but also an aggregated bucket that are useful for analytics dashboard for example.

For this project elasticsearch version 5.2.0 was used (latest at that time). The installation was performed via CLI and a Debian package.

The list of commands used for installation and configuration can be found in appendices.

During the installation, elasticsearch installer installs it to **/usr/share/elasticsearch**, configuration files are placed to **/etc/elasticsearch** and initialization script to **/etc/init.d/elasticsearch**. The instance was started as a background service. To ensure that elasticsearch starts and stops automatically with the server, initialization script was added to the default run level.

Elasticsearch configuration files are stored in **/etc/elasticsearch** and there are two types of them:

- **elasticsearch.yml** file is used to configure elasticsearch server settings
- **logging.yml** file stores logging configurations. By default, all logs are written to `/var/log/elasticsearch`

As by default it is running on localhost with the port 9200, it is possible to check instance status via REST API that elasticsearch provides but sending GET request to **http://localhost:9200/_cluster/health** :


```

damir@emqtt:/etc/init.d$ sudo service elasticsearch status
* elasticsearch is running
damir@emqtt:/etc/init.d$ curl -XGET "http://localhost:9200/_cluster/health?pretty=true"
{
  "cluster_name" : "elasticsearch",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 0,
  "active_shards" : 0,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "delayed_unassigned_shards" : 0,
  "number_of_pending_tasks" : 0,
  "number_of_in_flight_fetch" : 0,
  "task_max_waiting_in_queue_millis" : 0,
  "active_shards_percent_as_number" : 100.0
}
damir@emqtt:/etc/init.d$ █

```

Figure 11. Elasticsearch instance status

Elasticsearch status is “green” as seen on the screenshot, and in order to make the site available over the specified address, Nginx server was install and configured as a reverse proxy as well as the networking and firewall.



```

{
  "name": "w8pMJJb",
  "cluster_name": "elasticsearch",
  "cluster_uuid": "r2PX_nXERn63VGD000dD2Q",
  "version": {
    "number": "5.2.0",
    "build_hash": "24e05b9",
    "build_date": "2017-01-24T19:52:35.800Z",
    "build_snapshot": false,
    "lucene_version": "6.4.0"
  },
  "tagline": "You Know, for Search"
}

```

Figure 12. Elasticsearch instance status

5.2 Analytics dashboard

As all the pieces are ready, the next step is to visualize the analytics data that was generated. It was decided to build a separate application that contains an analytics dashboard and makes it as an independent component so that it can then be easily integrated to APInf when it is well tested independently. The dashboard was built

with same technologies that power APIInf except the charting library. This time the choice was made in favor of NDV3. Just like DC.js, NVD3 is also based on D3 in its core, and contains same features that it provides and stands for “Re-usable chart for d3.js”. The web application was built using NodeJS runtime and MeteorJS client-server side framework written on JavaScript. Based on Wikipedia (2017d, 2017e, 2017f).

For the initial setup of the environment and configuration, the existing MeteorJS boilerplate was used (<https://github.com/frenchbread/meteor-boilerplate>). Beyond that, a number of supporting libraries was also added to the project: MomentJS – which helps to manipulate and format the JavaScript Date object, Lodash which was used for mapping, parsing and manipulating with a dataset fetched from elasticsearch instance, and finally official elasticsearch library for JavaScript used for fetching the actual analytics data.

Due to MeteorJS’s framework architecture, it has a system called Methods – a remote procedure call (RPC) system which allows saving user input events that come from the client-side. Meteor Methods operate over DDP (Distributed Data Protocol) that was created specifically for MeteorJS Framework for this communication. Code-wise these methods are defined by passing a function to the “Meteor.methods({})” constructor that takes an object as a parameter, in a key-value pattern, where key is a name of the method to be called on client-side and the value is a function to be executed. This method was called `getAggregatedData`. Before actually writing a method, elasticsearch class instance must be initialized. `ElasticSearch.Client` class takes an object with an elasticsearch host value that points to elasticsearch instance. In general, the function should do following:

1. Take one argument as an object that is constructed and passed from the client-side and contains options for elasticsearch such as document index, number of records to be returned and a search query.
2. Run elasticsearch client command.
3. Receive data.

4. Parse data so that it is ready for NVD3 on the other end without further actions.
5. Return parsed data.
6. Display an error, if it has occurred.

On the client side the algorithm is simpler. First, it is needed to fetch the data and pass it to a function that will parse it. For the first step, in order to get data from elasticsearch using the method created earlier, it should be called once the page completes rendering. MeteorJS has a number of event listeners which are emitted when certain action finishes, for instance, when a template is created and when a template is rendered. The difference between these two callbacks is that “onRendered” gets fired when DOM elements finish rendering, while “onCreated” callback is fired when the template has been initialized but not yet rendered. To have the code clean and more readable, two functions were defined and attached to a template instance inside “onCreated” callback. This does not really affect the work flow but, can decrease page loading time and increase performance. Since the main code that runs these functions is placed inside “onRendered” callback, the functions for fetching data and rendering charts will be already defined. So, inside “onRendered” callback the pattern for calling the defined function will follow the following logic:

1. Call the “getAggregationData” method that fetches analytics data and returns a callback function with error and response arguments.
2. Check, if an error did occur while running the method and if it did, display it.
3. Render charts.

6 RESULTS

For the final version of the dashboard, in addition to the initial configuration and features, a number of filtering options were added. Similarly, to the APInf dashboard, one of them is granularity selection and date range filtering. Unlike in the APInf dashboard which shows the usage of HTTP request, the EMQ dashboard renders data which is generated over MQTT connections and as these protocols are different and, for instance, the MQTT protocol does not have status codes, it is not possible to display such kind of data. In fact, event types have been saved, and as seen in Figure 12, are represented in different colors making it easy to differentiate them from each other. The canvas that contains all the charts now has a number of them rendered at the same time. Specific color on the chart corresponds to the same color of the event listed above the canvas. This was implemented using the NVD3 feature, that allows to support multiple data set inputs. Dashboard is also filterable by event types, so when clicking on a colored circle that represents certain event type, the chart will instantly re-render to the selected charts minimum. Multi-event selections are supported.

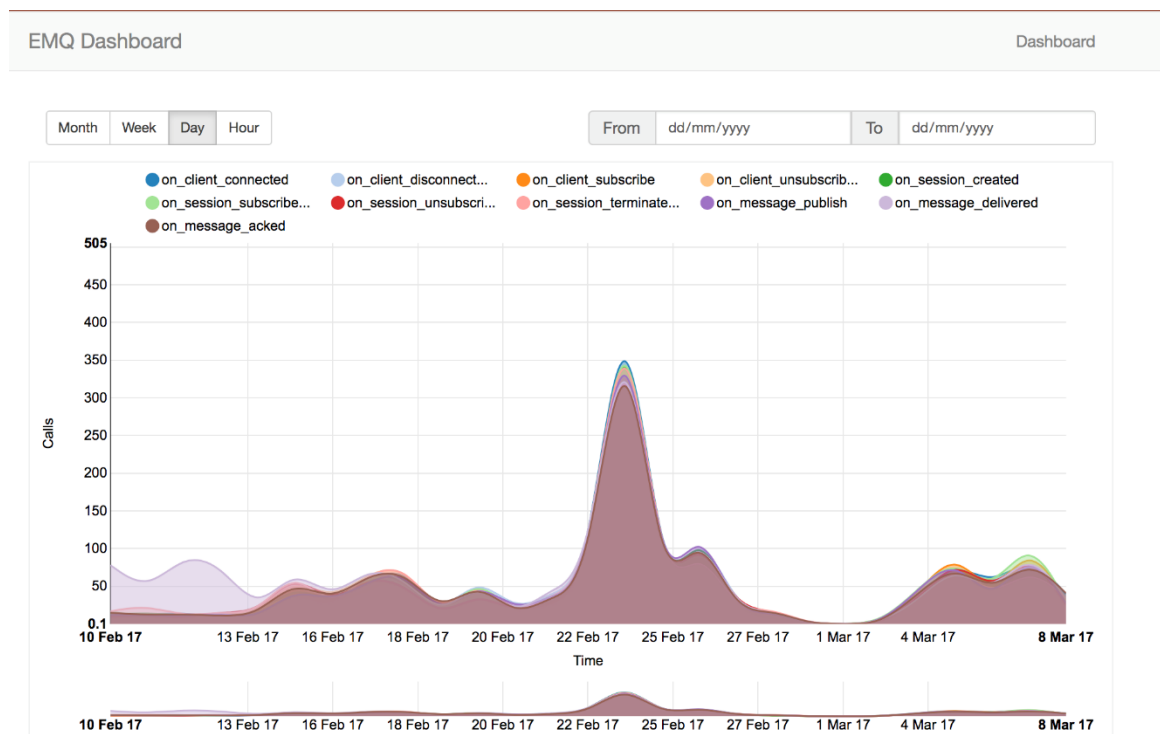


Figure 12. First public preview of the dashboard

As new suggestion and feature requests arrived, the dashboard got significant improvements as of the UX and the backend. Figure 13, shows that the event-type filtering has been redesigned and refactored towards selectbox, manually.

The feature set is still the same, but represented in a differently in more compact way. It is also worth mentioning that all the filter changes that are triggered (e.g. selected), instantly affect the chart, forcing it to re-render and update the URL by appending query a value to it. Modifying URL when filter changes, makes it easier to share the specific filter query with someone else, since the user vireing it on the other side will get the same rendered ranges for dates and other filters.

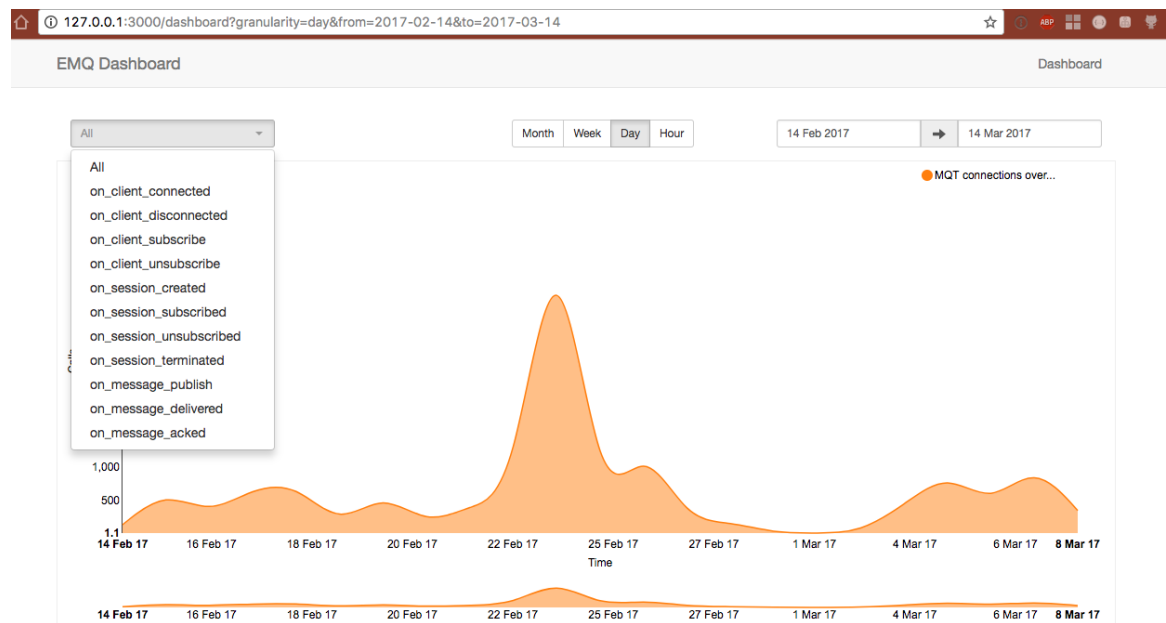


Figure 13. Dashboard final look

Performance is very important aspect of dashboards that provide analytics data on a scale. When the number of requests is growing, the system should be able to handle this, using common approached such as caching, indexing and load balancing. Performance audit has been done for the EMQ dashboard in order to find pit holes which introduce various memory leakages or useless code execution. Referring to Figure 14, it is possible to say that the complete page loading time takes around two seconds (2.200 seconds). This metric contains all the requests made to the elasticsearch including data aggregation. In fact, the

rendering takes around (0.300 seconds). This metric is collected over one month date range and contains around 3000 items in the data set.

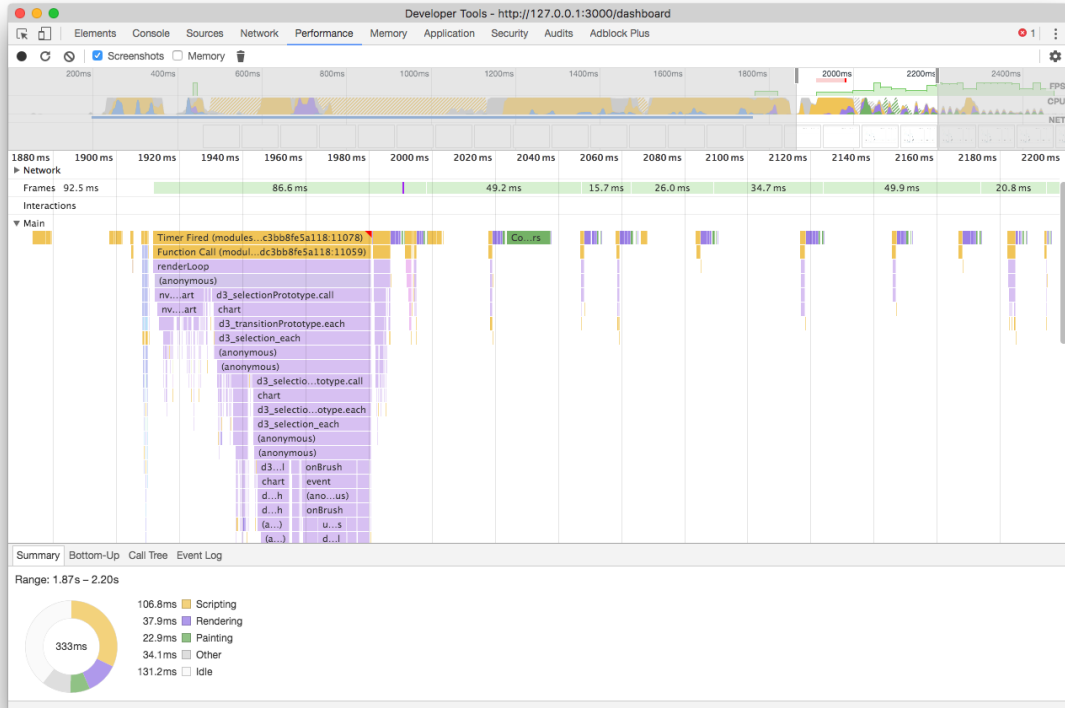


Figure 14. Performance metrics from Google Chrome Dev tools

In order to measure the performance with a wider date-range, a bunch of dummy data has been generated.

Table 11. Performance metrics for larger data-set

Items count	Server -> ES request (ms)	ES search execution (ms)
13k	42.9	3.9
20k	30	3.7
50k	28.7	2.3

Table 11 represents average values for the metrics provided in Appendix 6. First column contains the number of items stored in the elasticsearch and from which aggregated data-set has been generated for the chart. The column in the middle shows the time which takes the server to request data from elasticsearch in milliseconds. The third column shows the amount of time which it takes the elasticsearch to aggregate the data. The metrics in the third column have been

collected manually while metrics in third column is exposed in the elasticsearch response object. Therefore, the more entities in the elasticsearch datastore, the faster execution and aggregation time.

7 CONCLUSIONS AND FUTURE WORK

I must say that the task given me by the company was quite challenging, even though it sounded easy at first. It was challenging in a degree that required to cover multiple IT fields at once and at some point to make a step in creating an ecosystem for Internet of Things. I, as a new-comer to the Erlang world, had to learn a new programming language which involved research and writing sample programs, to be able to understand what is going on under the hood of EMQ and to extend it with a plugin. The vast majority of the software that was used, is open-sourced and is driven by the community.

As of the work within this thesis is finished, the work on company's side is ongoing and heading towards having complete multi-proxy architecture. That is why, It is required to add support for features provided by EMQ such as access control rules and user management. The EMQ analytics dashboard that was created needs to be integrated into APIInf with possible context selection for either HTTP or MQTT supported dataset. During the dashboard development, the plugin has been renamed to emq-plugin-elasticsearch, a more generalized version, that is going to be contributed to EMQ upstream repository.

REFERENCES

Erlang programming language 2017a. Wikipedia. WWW document. Available at: [https://en.wikipedia.org/wiki/Erlang_\(programming_language\)](https://en.wikipedia.org/wiki/Erlang_(programming_language)) [Accessed 20 February 2017]

Hypertext Transfer Protocol 2017b. Wikipedia. WWW document. Available at: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol [Accessed 29 March 2017]

MQTT 2017c. Wikipedia. WWW document. Available at: <https://en.wikipedia.org/wiki/MQTT> [Accessed 29 March 2017]

JavaScript 2017d. Wikipedia. WWW document. Available at: <https://en.wikipedia.org/wiki/JavaScript> [Accessed 20 February 2017]

Node.js 2017e. Wikipedia. WWW document. Available at: <https://en.wikipedia.org/wiki/Node.js> [Accessed 25 February 2017]

Meteor (web framework) 2017f. Wikipedia. WWW document. Available at: [https://en.wikipedia.org/wiki/Meteor_\(web_framework\)](https://en.wikipedia.org/wiki/Meteor_(web_framework)) [Accessed 25 February 2017]

Erlang MQTT Broker website 2017. EMQ 2.0 documentation. WWW document. Available at: <http://emqtt-docs.readthedocs.io/en/emq20/design.html#design-philosophy> [Accessed 17 March 2017]

Fred Hebert. 2013. Learn You Some Erlang for Great Good, A Beginner Guide. Available at: <http://learnyousomeerlang.com/> [Accessed 13 February 2017]

Joe Armstrong. A History of Erlang, Ericsson AB. Available at: http://webcem01.cem.itesm.mx:8005/erlang/cd/downloads/hopl_erlang.pdf [Accessed 20 March 2017]

APPENDICES

1. Elasticsearch installation commands

```
# Fetch elasticsearch debian package from the official source
$ wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-5.2.0.deb

# Install it as a debian package
$ sudo dpkg -I elasticsearch-5.2.0.deb

# Install JDK 8
$ sudo apt-get install default-jre
$ sudo apt-get install default-jdk

$ sudo apt-get install oracle-java8-installer
$ add-apt-repository ppa:webupd8team/java
$ sudo apt-get update

# Update /etc/environment file by settings JAVA_HOME variable
JAVA_HOME="/usr/lib/jvm/java-8-oracle"

# Reload the environment
$ source /etc/environment

# Start elasticsearch as a service
$ sudo service elasticsearch start
```

2. NGINX proxy configuration

```
server {
    listen 9200;

    server_name <IP_ADDRESS>;

    location / {
        proxy_pass http://localhost:9200;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}

# Also update linux firewall by allowing port 9200 to pass.
# Reload nginx service with:
$ sudo service nginx reload
```

3. Server-side code (emq-analytics-dashboard)


```

// The full repository is available on github:
https://github.com/apinf/emq-analytics-dashboard

import { Meteor } from 'meteor/meteor';
import ES from 'elasticsearch';
import _ from 'lodash';

import parseDataForNvd from './lib/parse';
import config from '../../config';

// Initialize ES instance
const client = new ES.Client({
  host: config.host,
});

const opts = {
  // Initial elasticsearch index & type values
  index: 'mqtt',
  type: 'events',

  // Amount of items to return
  // Since aggregated data is returned, regular records are not needed
  size: 0,
};

Meteor.methods({
  getChartData (query) {
    // Merge default query with custom
    const esOpts = _.assign(opts, query);

    // Execute search
    return client.search(esOpts).then(
      (res) => {
        const data = res.aggregations.logs_over_time.buckets;
        return parseDataForNvd(data);
      },
      (err) => { return new Meteor.Error(err); }
    );
  },
  getEventTypes () {
    // Construct custom query
    const query = {
      body: {
        aggs: {
          types: {
            terms: {
              field: 'event',
            },
          },
        },
      },
    };

    // Merge default query with custom
    const esOpts = _.assign(opts, query);

    // Execute search
    return client.search(esOpts).then(
      (res) => { return res.aggregations.types.buckets; },
      (err) => { return new Meteor.Error(err); }
    );
  }
});

```

```

    );
  },
  getTopics () {
    // Construct custom query
    const query = {
      body: {
        aggs: {
          topics: {
            terms: {
              field: 'topic',
            },
          },
        },
      },
    };
    // Merge default query with custom
    const esOpts = _.assign(opts, query);

    // Execute search
    return client.search(esOpts).then(
      (res) => { return res.aggregations.topics.buckets; },
      (err) => { return new Meteor.Error(err); }
    );
  },
});

```

4. Client-side code (emq-analytics-dashboard)

```

// The full repository is available on github:
https://github.com/apinf/emq-analytics-dashboard

import { Meteor } from 'meteor/meteor';
import { Template } from 'meteor/templating';
import { FlowRouter } from 'meteor/kadira:flow-router';

import nvd3 from 'nvd3';
import d3 from 'd3';
import moment from 'moment';
import _ from 'lodash';

Template.dashboard.onCreated(function () {
  const instance = this;

  instance.opts = {
    body: {
      query: {
        bool: {
          must: [],
        },
      },
    },
    aggs: {
      logs_over_time: {
        date_histogram: {
          field: 'timestamp',
          interval: '',
          format: 'dd-MM-yyyy',
        },
      },
    },
  },

```

```

    },
  },
};

instance.getChartData = (opts) => {
  return new Promise((resolve, reject) => {
    Meteor.call('getChartData', opts, (err, res) => {
      if (err) reject(err);
      resolve(res);
    });
  });
};

instance.render = (data) => {
  if (data) {
    nv.addGraph(() => {
      const chart = nv.models.lineWithFocusChart();

      const tickMultiFormat = d3.time.format.multi([
        ['%-d %b %y', (d) => { return d.getDate(); }],
        ['%b %-d', (d) => { return d.getMonth(); }],
        ['%Y', () => { return true; }],
      ]);

      chart.interpolate('basis');

      chart.xAxis
        .tickFormat((d) => { return tickMultiFormat(new Date(d)); })
        .axisLabel('Time');

      chart.x2Axis
        .tickFormat((d) => { return tickMultiFormat(new Date(d)); })
        .axisLabel('Overview chart');

      chart.yAxis
        .tickFormat(d3.format(',.2'))
        .axisLabel('Calls');

      chart.y2Axis
        .tickFormat(d3.format(',.2'));

      chart.margin({ left: 70, right: 40 });

      d3.select('#chart svg')
        .attr('height', 500)
        .datum(data)
        .transition()
        .duration(500)
        .call(chart);

      nv.utils.windowResize(chart.update);

      return chart;
    });
  }
};

instance.init = (opts) => {
  instance.getChartData(opts)
    .then((items) => {
      instance.render(items);
    });
};

```

```

    })
    .catch((err) => { return console.error(err); });
  });

  instance.updateQuery = () => {
    const from = FlowRouter.getQueryParam('from') || moment().subtract(1, 'month').format('YYYY-MM-DD');
    const to = FlowRouter.getQueryParam('to') || moment().format('YYYY-MM-DD');
    const granularity = FlowRouter.getQueryParam('granularity') || 'day';
    const emqEvent = FlowRouter.getQueryParam('event') || '';
    const topic = FlowRouter.getQueryParam('topic') || '';

    const mustQuery = instance.opts.body.query.bool.must;

    // Find & remove "range" query object from array
    // so that we can update query with "fresh" rules
    const range = _.find(mustQuery, (obj) => { return typeof obj.range !== 'undefined'; });
    if (range) {
      _.remove(mustQuery, _.find(mustQuery, (obj) => { return typeof obj.range !== 'undefined'; }));
    }

    // Find & remove "match" query object from array
    // so that we can update query with "fresh" rules
    const match = _.find(mustQuery, (obj) => { return typeof obj.match !== 'undefined'; });
    if (match) {
      _.remove(
        mustQuery,
        (q) => { return typeof q.match === 'object'; }
      );
    }

    // Push "filter-by-event" query
    if (emqEvent) {
      mustQuery.push({ match: { event: emqEvent } });
    }

    // Push "filter-by-topic" query
    if (topic) {
      mustQuery.push({ match: { topic } });
    }

    // Push "filter-by-range" query
    mustQuery.push({
      range: {
        timestamp: {
          gte: from,
          lte: to,
          format: 'yyyy-MM-dd',
        },
      },
    });

    instance.opts.body.query.bool.must = mustQuery;
    instance.opts.body.aggs.logs_over_time.date_histogram.interval = granularity;
  });

```

```
});

Template.dashboard.onRendered(function () {
  const instance = this;

  instance.autorun(() => {
    instance.updateQuery();
    instance.init(instance.opts);
  });
});
});
```

5. Erlang code for a plugin (emqtt_plugin_apinf)

```
// The full repository is available on github:
https://github.com/apinf/emq-plugin-apinf/

-module(emq_plugin_apinf).

-include_lib("emqtt/include/emqtt.hrl").

-export([load/1, unload/0]).

%% Hooks functions

-export([on_client_connected/3, on_client_disconnected/3]).
-export([on_client_subscribe/4, on_client_unsubscribe/4]).

-export([on_session_created/3, on_session_subscribed/4, on_session_un-
subscribed/4, on_session_terminated/4]).

-export([on_message_publish/2, on_message_delivered/4, on_mes-
sage_acked/4]).

write_to_es(Log) ->
  esio:start(),
  % TODO: Move ES host URL to config file
  {ok, Sock} = esio:socket("http://192.168.43.171:9200/"),
  Id = uuid:to_string(uuid:uuid1()),
  esio:put(Sock, "urn:es:mqt:analytics:" ++ Id, Log),
  esio:close(Sock).

% --- Custom functions

%% Called when the plugin application start
load(Env) ->
  emqtt:hook('client.connected', fun ?MODULE:on_client_connected/3,
[Env]),
  emqtt:hook('client.disconnected', fun ?MODULE:on_client_discon-
nected/3, [Env]),
  emqtt:hook('client.subscribe', fun ?MODULE:on_client_subscribe/4,
[Env]),
  emqtt:hook('client.unsubscribe', fun ?MODULE:on_client_unsub-
scribe/4, [Env]),
  emqtt:hook('session.created', fun ?MODULE:on_session_created/3,
[Env]),
  emqtt:hook('session.subscribed', fun ?MODULE:on_session_sub-
scribed/4, [Env]),
```

```

    emqttd:hook('session.unsubscribed', fun ?MODULE:on_session_unsub-
scribed/4, [Env]),
    emqttd:hook('session.terminated', fun ?MODULE:on_session_termi-
nated/4, [Env]),
    emqttd:hook('message.publish', fun ?MODULE:on_message_publish/2,
[Env]),
    emqttd:hook('message.delivered', fun ?MODULE:on_message_deliv-
ered/4, [Env]),
    emqttd:hook('message.acked', fun ?MODULE:on_message_acked/4,
[Env]).

on_client_connected(ConnAck, Client = #mqtt_client{client_id = Client-
Id}, _Env) ->
    io:format("client ~s connected, connack: ~w~n", [ClientId, Conn-
Ack]),
    Log = #{
        type => <<"on_client_connected">>,
        date => erlang:localtime()
    },
    write_to_es(Log),
    {ok, Client}.

on_client_disconnected(Reason, _Client = #mqtt_client{client_id = Cli-
entId}, _Env) ->
    io:format("client ~s disconnected, reason: ~w~n", [ClientId, Rea-
son]),
    Log = #{
        type => <<"on_client_disconnected">>,
        date => erlang:localtime()
    },
    write_to_es(Log),
    ok.

on_client_subscribe(ClientId, Username, TopicTable, _Env) ->
    io:format("client(~s/~s) will subscribe: ~p~n", [Username, Client-
Id, TopicTable]),
    Log = #{
        type => <<"on_client_subscribe">>,
        date => erlang:localtime(),
        username => Username,
        topic_table => TopicTable
    },
    write_to_es(Log),
    {ok, TopicTable}.

on_client_unsubscribe(ClientId, Username, TopicTable, _Env) ->
    io:format("client(~s/~s) unsubscribe ~p~n", [ClientId, Username,
TopicTable]),
    Log = #{
        type => <<"on_client_unsubscribe">>,
        date => erlang:localtime(),
        username => Username,
        topic_table => TopicTable
    },
    write_to_es(Log),
    {ok, TopicTable}.

on_session_created(ClientId, Username, _Env) ->
    Log = #{
        type => <<"on_session_created">>,
        date => erlang:localtime(),

```

```

    username => Username
  },
  write_to_es(Log),
  io:format("session(~s/~s) created.", [ClientId, Username]).

on_session_subscribed(ClientId, Username, {Topic, Opts}, _Env) ->
  io:format("session(~s/~s) subscribed: ~p~n", [Username, ClientId,
{Topic, Opts}]),
  Log = #{
    type => <<"on_session_subscribed">>,
    date => erlang:localtime(),
    username => Username,
    topic_and_opts => #{
      topic => Topic,
      opts => Opts
    }
  },
  write_to_es(Log),
  {ok, {Topic, Opts}}.

on_session_unsubscribed(ClientId, Username, {Topic, Opts}, _Env) ->
  io:format("session(~s/~s) unsubscribed: ~p~n", [Username, Client-
tId, {Topic, Opts}]),
  Log = #{
    type => <<"on_session_unsubscribed">>,
    date => erlang:localtime(),
    username => Username,
    topic_and_opts => #{
      topic => Topic,
      opts => Opts
    }
  },
  write_to_es(Log),
  ok.

on_session_terminated(ClientId, Username, Reason, _Env) ->
  io:format("session(~s/~s) terminated: ~p.", [ClientId, Username,
Reason]),
  Log = #{
    type => <<"on_session_terminated">>,
    date => erlang:localtime(),
    username => Username,
    reason => Reason
  },
  write_to_es(Log).

%% transform message and return
on_message_publish(Message = #mqtt_message{topic = <<"$SYS/", _/bi-
nary>>}, _Env) ->
  {ok, Message};

on_message_publish(Message, _Env) ->
  io:format("publish ~s~n", [emqtt_message:format(Message)]),
  #mqtt_message{
    from = {_, UsernameFrom},
    qos = Qos,
    retain = Retain,
    dup = Dup,
    topic = Topic
  } = Message,
  Log = #{

```

```

type => <<"on_message_publish">>,
date => erlang:localtime(),
message => #{
    from => UsernameFrom,
    qos => Qos,
    retain => Retain,
    topic => Topic,
    dup => Dup
}
},
write_to_es(Log),
{ok, Message}.

on_message_delivered(ClientId, Username, Message, _Env) ->
    io:format("delivered to client(~s/~s): ~s~n", [Username, ClientId,
emqtt_message:format(Message)]),
    #mqtt_message{
        from = {_, UsernameFrom},
        qos = Qos,
        retain = Retain,
        dup = Dup,
        topic = Topic
    } = Message,
    Log = #{
        type => <<"on_message_delivered">>,
        date => erlang:localtime(),
        username => Username,
        message => #{
            from => UsernameFrom,
            qos => Qos,
            retain => Retain,
            topic => Topic,
            dup => Dup
        }
    },
    write_to_es(Log),
    {ok, Message}.

on_message_acked(ClientId, Username, Message, _Env) ->
    io:format("client(~s/~s) acked: ~s~n", [Username, ClientId,
emqtt_message:format(Message)]),
    #mqtt_message{
        from = {_, UsernameFrom},
        qos = Qos,
        retain = Retain,
        dup = Dup,
        topic = Topic
    } = Message,
    Log = #{
        type => <<"on_message_acked">>,
        date => erlang:localtime(),
        username => Username,
        message => #{
            from => UsernameFrom,
            qos => Qos,
            retain => Retain,
            topic => Topic,
            dup => Dup
        }
    },
    write_to_es(Log),

```



```

    {ok, Message}.

%% Called when the plugin application stop
unload() ->
    emqttd:unhook('client.connected', fun ?MODULE:on_client_connected/3),
    emqttd:unhook('client.disconnected', fun ?MODULE:on_client_disconnected/3),
    emqttd:unhook('client.subscribe', fun ?MODULE:on_client_subscribe/4),
    emqttd:unhook('client.unsubscribe', fun ?MODULE:on_client_unsubscribe/4),
    emqttd:unhook('session.subscribed', fun ?MODULE:on_session_subscribed/4),
    emqttd:unhook('session.unsubscribed', fun ?MODULE:on_session_unsubscribed/4),
    emqttd:unhook('message.publish', fun ?MODULE:on_message_publish/2),
    emqttd:unhook('message.delivered', fun ?MODULE:on_message_delivered/4),
    emqttd:unhook('message.acked', fun ?MODULE:on_message_acked/4).

```

6. Request/Aggregation benchmarks

Items count	Days amount	Server → ES (ms)	ES search exec (ms)
13003	66	92	4
13003	66	75	6
13003	66	49	5
13003	66	25	3
13003	66	22	3
13003	66	24	3
13003	66	22	3
13003	66	28	2
13003	66	49	6
Average		42.88888888888889	3.888888888888889
20626	66	53	7
20626	66	32	2
20626	66	20	2
20626	66	27	6
20626	66	23	3
20626	66	45	5
20626	66	26	3
20626	66	23	2
20626	66	29	4
20626	66	22	3
Average		30	3.7
50009	66	65	2
50009	66	33	2
50009	66	28	2
50009	66	23	2
50009	66	22	2
50009	66	22	2
50009	66	24	4
50009	66	21	3
50009	66	26	2

50009	66	24	2
Average		28.8	2.3