

A Client Interface for interacting with a privacy-preserving IoT middleware

MASTER THESIS
SWISS JOINT MASTER OF SCIENCE IN COMPUTER SCIENCE
UNIVERSITY OF BERN, NEUCHÂTEL AND FRIBOURG

MARCEL GROSJEAN

December 2018

Thesis supervisors:

Prof. Dr. Jacques PASQUIER–ROCHA
Software Engineering Group

Pascal Gremaud
Software Engineering Group

“If you don’t do something because you think you can’t do it, you’ll never be able to do anything in the future.”

- *Kira Yamato, Mobile Suit Gundam Seed*

Abstract

This work is part of the research project for data confidentiality in IoT led by the University of Fribourg, Software Engineering Group. It represents the intersection of the domains of the Internet of Things, software security as well as software engineering.

In the context of research in the IoT, the Softeng group is developing a privacy-preserving IoT middleware using Trusted Execution Environment, namely Intel Software Guard Extensions (SGX). This middleware is capable of interacting with smart devices and clients while all the sensitive data are hidden from the platform hypervisor, enabling trusted computing in untrusted environment [1]. In this work we have studied the different existing technologies in order to design a web application for middleware management. The interface was called RIOT (Responsive Internet of Things) and offers all the features to manage a middleware and offers in addition a completely secure communication. RIOT implements the ECA (event, condition and action) programming paradigm.

To run the front end, it was necessary to develop all the infrastructure around it in order to simulate the final environment in which it will be executed. Vue.js was chosen for the development of the interface. Encryption is necessary, so we decided to make it as transparent as possible to the developer. Spring Boot was used to develop the middleware and a software actuator was develop with NodeJS. The rule engine was also developed in order to dynamically map rules to events and triggers actions on our software actuator.

Keywords: IoT; Middleware; Privacy-Aware computing; Security; Trusted Execution Environments; Intel SGX; Web interface; Vue.js; Web Crypto API; Rule engine; Zuul; iFlux; Material Design; Design Science

Acknowledgements

I want to thank all the people who helped me, guided and supported me during the realization of my master thesis. These people allowed me to conduct my research work until completion and helped me to overcome the difficulties encountered throughout the development of this work.

First of all, I would like to thank Professor Dr. Jacques Pasquier-Rocha because without him the realization of this research project would not have been possible.

My gratefully thanks also go to Pascal Gremaud for his kindness, his patience and his availability which allowed me to better understand the project's context. He was able to guide me during the realization of this software artifact. He was a mentor throughout the project and he was able to leave me free of my choices while putting me back on track during my periods of doubt and misguidance.

Table of Contents

1 Introduction	12
1.1 Context	13
1.2 Project Objectives.....	13
1.3 Issues and Needs	13
1.4 Global Approach	14
1.5 Thesis Structure.....	14
1.6 Writing Conventions	14
2 Theoretical foundations	15
2.1 Introduction	16
2.2 Design Science Research Methodology	16
2.3 The Internet of Things (IoT)	17
2.4 Project Context	18
2.5 Project Objectives.....	19
2.6 The iFlux project	20
2.6.1 Introduction.....	20
2.6.2 Smart Cities.....	21
2.6.3 The iFlux programming model	21
2.7 Trusted Execution Environment (TEE).....	25
2.7.1 Introduction.....	25
2.7.2 Intel SGX	26
2.8 The IoT middleware	27
2.9 Communication protocol.....	28
2.9.1 Introduction.....	28
2.9.2 Key exchange algorithms.....	29
2.9.3 Advanced Encryption Standard (AES)	31
2.9.4 Message exchange protocol	32
2.10 Conclusion.....	36
3 Design and Implementation	37
3.1 Introduction	39
3.2 Risks Analysis	39
3.3 Global Architecture	41

3.4	Cryptography middleware.....	42
3.4.1	Introduction.....	42
3.4.2	Web Crypto API	42
3.4.3	Java Security	44
3.4.4	Benchmark	44
3.4.5	Spring Zuul	47
3.5	Custom REST API	49
3.5.1	Introduction.....	49
3.5.2	Keys	49
3.5.3	Users	50
3.5.4	Clients	51
3.5.5	Auth.....	52
3.5.6	Client Urls.....	52
3.5.7	Event types.....	53
3.5.8	Action types	54
3.5.9	Rules	55
3.5.10	Event	57
3.5.11	Conclusion	58
3.6	Enclave middleware	59
3.6.1	Introduction.....	59
3.6.2	Spring boot.....	59
3.6.3	Authentication/Authorization	62
3.6.4	Errors Handling & Logging.....	64
3.6.5	Database type & model.....	66
3.6.6	Database Access.....	68
3.6.7	Rule engine	70
3.7	Action Target.....	78
3.8	Front-end (RIOT)	80
3.8.1	Introduction.....	80
3.8.2	Use cases	82
3.8.3	Navigation Diagram.....	83
3.9	Practical example of cryptographic functions	85
3.9.1	Introduction.....	85
3.9.2	Session Key Creation.....	85
3.9.3	Authentication.....	90
3.9.4	Message exchange	94
3.10	Software development organization.....	95
3.10.1	Source code version control.....	95

3.10.2 Iterative Development.....	95
3.10.3 Testing.....	96
3.10.4 Deployment.....	97
3.11 Conclusion.....	99
3.12 Improvements & Future Works.....	100
4 Practical testing and evaluation	101
4.1 Introduction	102
4.2 RIOT scenario	103
4.2.1 Introduction.....	103
4.2.2 Login	104
4.2.3 Verify session key.....	105
4.2.4 Users	107
4.2.5 Clients	107
4.2.6 Urls (action targets urls).....	108
4.2.7 Event type	109
4.2.8 Action type.....	110
4.2.9 Rules	111
4.2.10 Graph explorer	114
4.3 Client scenario.....	115
4.3.1 Introduction.....	115
4.3.2 Client.....	116
4.3.3 Emails sent.....	118
4.3.4 Events List	119
4.3.5 Action message	120
4.4 Conclusion.....	120
5 Administrative part	121
5.1 Introduction	122
5.2 Planification.....	122
5.3 Logbook.....	123
5.4 Burndown chart	128
5.5 Bibliography & Webography	129
6 Appendix	133
6.1 Cryptography benchmark Readme.....	134
6.2 Front-end Readme	140
6.3 Enclave Readme	142
6.4 Zuul Gateway Readme	147
6.5 Email sender Readme	151

List of Figures

Figure 1: The secure middleware architecture. Illustration from P. Gremaud [1].....	19
Figure 2: iFlux components representation.....	24
Figure 3: Intel SGX enclave, trusted/untrusted parts.....	26
Figure 4: General architecture of the system. Illustration by P. Gremaud [1].....	27
Figure 5: Graphical example of the AES algorithm	31
Figure 6: RIOT global architecture.....	41
Figure 7: Web cryptography benchmark. Native vs JS implementation	43
Figure 8: Gateway source code.....	48
Figure 9: Keys API illustration	50
Figure 10: Users API illustration	50
Figure 11: Clients API illustration.....	51
Figure 12: Auth API illustration	52
Figure 13: Urls API illustration	53
Figure 14: Event type API illustration.....	54
Figure 15: Action type API illustration	55
Figure 16: Rules API illustration	57
Figure 17: Events API illustration	58
Figure 18: Enclave project structure part 1.....	60
Figure 19: Enclave project structure part 2.....	61
Figure 20: Logic data model.....	67
Figure 21: Rule engine schematic illustration	70
Figure 22: Action target API illustration	78
Figure 23: Front-end Framework ranking 2018.....	80
Figure 24: RIOT use cases diagram.....	82
Figure 25: RIOT navigation diagram.....	83
Figure 26: Front-end project structure	84
Figure 27: Session establishment sequence	85
Figure 28: Authentication sequence.....	90
Figure 29: Message exchange sequence	94

Figure 30: Login page.....	104
Figure 31: Home page.....	104
Figure 32: User configuration page	105
Figure 33: Keys list page	105
Figure 34: Session key details page.....	106
Figure 35: User details page for the administrator.....	107
Figure 36: Clients list page	107
Figure 37: Clients details page.....	108
Figure 38: Urls list page.....	108
Figure 39: Url details page.....	109
Figure 40: Event types list page.....	109
Figure 41: Event type details page.....	110
Figure 42: Action type list page.....	110
Figure 43: Action type details page	111
Figure 44: Rules list page	111
Figure 45: Rule details page	112
Figure 46: Rule action page.....	113
Figure 47: Graph explorer.....	114
Figure 48: Client left part.....	116
Figure 49: Client right part	117
Figure 50: Mailbox after actions triggered	118
Figure 51: Logging email.....	118
Figure 52: Detailed email.....	119
Figure 53: Events list page.....	119
Figure 54: Action message page	120
Figure 55: Burndown chart	128

List of Tables

Table 1: Design Science Research Methodology [2].....	16
Table 2: Diffie-Hellman private key exchange steps.....	29
Table 3: Example of Diffie-Hellman with numbers	30
Table 4: List of risks	39
Table 5: Risks matrix	40
Table 6: Java vs Web Crypto API scenario	45
Table 7: GANTT planification.....	122
Table 8: Logbook.....	123

List of source code

Code 1: iFlux event.....	22
Code 2: iFlux action target.....	23
Code 3: iFlux rules.....	24
Code 4: Example of session creation.....	33
Code 5: JWE as JWT compact serialization example	33
Code 6: JWE header.....	33
Code 7: JWE JSON example	34
Code 8: JWS as JWT example.....	34
Code 9: JWS header.....	35
Code 10: JWS payload.....	35
Code 11: Authorization header	35
Code 12: Session creation.....	49
Code 13: User creation.....	50
Code 14: Client creation	51
Code 15: User authentication.....	52
Code 16: Url creation.....	52
Code 17: Event type creation.....	53
Code 18: Action type creation	54
Code 19: Rule creation.....	55
Code 20: Rule action creation.....	56
Code 21: Event fired	57
Code 22: GET action messages	58
Code 23: JWT example after authentication.....	62
Code 24: Servlet Filter example	62
Code 25: Example or protected route	63
Code 26: Role enum.....	63
Code 27: Spring Boot annotations handler	64
Code 28: Spring Boot exceptions handler	65
Code 29: Example of error returned by the API	66

Code 30: DAO example.....	69
Code 31: KeyRow mapper.....	69
Code 32: Send event for rule engine.....	72
Code 33: Event type example for rule engine.....	72
Code 34: Actions for specific client and event type	73
Code 35: Basic string for evaluation.....	73
Code 36: Script that will trigger an action	73
Code 37: Actions of a rule	74
Code 38: Action type of a rule's action	74
Code 39: Turn String JSON into Java JSON Object	74
Code 40: Turn String scheme into a Java Map	75
Code 41: Data transformations done by the rule engine.....	76
Code 42: Example of templating	76
Code 43: Templating done.....	76
Code 44: Data transformation on numerical values.....	77
Code 45: Final action message	77
Code 46: Action target example with NodeJS.....	79
Code 47: Example of action message for sending an email	79
Code 48: Example of session creation	86
Code 49: Generate a key pair from Java.....	86
Code 50: Generate shared secret in Java.....	87
Code 51: Example of ciphered payload returned by the enclave.....	87
Code 52: Creation of a shared secret on the client side	88
Code 53: Shared secret creation and derivation with web crypto API	89
Code 54: Intercept HTTP request with Axios.....	91
Code 55: Decryption of request payload by Zuul.....	92
Code 56: Gateway automated tests	96
Code 57: Email sender CURL test.....	96
Code 58: Enclave automated tests	97

Acronyms

AES-GCM	Advanced Encryption Standard, Galois/Counter Mode
API	Application programming Interface
CSS	Cascading Style Sheet
DOM	Document Object Model
ECA	Events Condition Action
ECDH	Elliptic Curve Diffie Hellman
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JS	JavaScript
JSON	JavaScript Object Notation
MVC	Model View Controller
NOSQL	Not Only SQL
POJO	Plain Old Java Object
REST	Representational State Transfer
SGX	Software Guard Extensions
SQL	Structured Query Language
TEE	Trusted Execution Enviroments
W3C	World Wide Web Consortium

1

Introduction

1.1 Context	13
1.2 Project Objectives	13
1.3 Issues and Needs	13
1.4 Global Approach	14
1.5 Thesis Structure	14
1.6 Writing conventions	14

1.1 Context

In the context of research in the Internet of Things, the Software Engineering Group of the University of Fribourg is developing a privacy-preserving middleware using Trusted Execution Environments (TEEs), namely Intel Software Guard Extensions (SGX). This middleware is capable of interacting with smart devices (Things) and clients, while all the application data are hidden from the platform hypervisor, enabling trusted computing in untrusted environments. In particular in cloud computing.

At its current state, the system proposes a REST interface, using symmetric encryption to safely transmit HTTP (JSON) message with both Things and clients. The middleware receives sensors events, and depending on given dynamic rules, is able to trigger actions to either Things or clients. The middleware assumes the platform owner is completely untrusted [1].

1.2 Project Objectives

The main goal of the thesis is to develop a web interface to interact with the backend middleware. We will call this backend an « enclave ». Because the context of the project is data privacy in IoT, the interface should not store any data on the server hosting the interface [1], making it a « trusted » component of our framework. For this same reason, JavaScript is preferred as the main programming language. The Web Crypto API offers a native implementation of most existing encryption standards and will provide cryptographic capabilities to the interface.

The web interface should act as an administration tool able to manage the enclave middleware.

All communications exchanges with the enclave middleware are done using encrypted messages packed in a JSON format.

1.3 Issues and Needs

The middleware project is currently under development and will be a server implementing a version derived from the iFlux project to manage data from Things. The management of actions, rules and events does not exist for the moment and must be done manually with CURL requests, for example.

A web application would provide a simpler and faster way to enter actions, rules and events so that events coming from Things can be managed more efficiently.

1.4 Global Approach

This master thesis requires research for information and continuous technical learning. It is necessary to understand and analyze the iFlux project as well as standard cryptographic methods if we want to develop the software.

First, how the different cryptographic methods that are ECDH and AES-GCM work had to be understood. Then, the iFlux project had to be assimilated.

Secondly, the technical and technological side had to be defined. The technical architecture of the project as well as the different use cases of the application was the result of the preliminary analysis. A preliminary learning process was necessary in order to assimilate the different technologies.

Finally, the application was developed by successive refining and each step was validated.

1.5 Thesis Structure

This document is structured in six sections. The first section is the introduction and introduces the global context. The second section introduces the important theoretical concepts for developing the complete architecture. It theoretically shows how and why it is necessary to develop an artifact. The third section is the practical part. It shows how the project was developed according to the theoretical section. The technological choices as well as the overall design are presented in this section. The fourth section shows a practical application by presenting a complete scenario. This scenario allows us to measure the relevance of the project as well as the technical possibilities related to the project developed. The fifth section presents the administrative part of the project. Planning, working hours and bibliography are included here. Finally, the last and sixth section contains appendices describing how to deploy the artifacts created during this project.

1.6 Writing conventions

This document contains several writing conventions. All examples of source code, tables and images are indexed. All references to them in text have a relative reference. Variables and important text parts are in italic in the text. The source code is numbered throughout the document. Any reference to an external author can be found in the administrative section.

2

Theoretical foundations

2.1 Introduction	16
2.2 Design Science Research methodology	16
2.3 The Internet of Things (IoT)	17
2.4 Project context	18
2.5 Project objectives	19
2.6 The iFlux project	20
2.6.1 Introduction	20
2.6.2 Smart cities	21
2.6.3 The iFlux programming model.....	21
2.7 TEEs	25
2.7.1 Introduction	25
2.7.2 Intel SGX.....	26
2.8 The IoT middleware	27
2.9 Communication protocol	28
2.9.1 Introduction	28
2.9.2 Key exchange algorithms	29
2.9.3 Advanced Encryption Standard (AES).....	31
2.9.4 Message exchange protocol.....	32
2.10 Conclusion	36

2.1 Introduction

This is the theoretical part where we will review the technical and theoretical constraints. We will see an overview of the different classical cryptographic techniques as well as how to implement our own version of the API based on the iFlux project. We will also see which technologies have been selected and why, in order to develop the prototype. Architectural design, technical implementation and tests are done in the further chapters.

2.2 Design Science Research Methodology

Design Science Research is an iterative development methodology and a set of techniques for conducting research in the field of Information Systems. It involves the creation of innovative artifacts (final product in the form of models or software) that advance the scientific field to which the project is linked [2].

The design process is a sequence of activities aimed at producing an innovative artifact. The artifact allows the researcher to have a better understanding of the problem, and an iterative process allows a re-evaluation of the work to arrive at a new result that better match expectations. The initial goal is to put forward all the theoretical concepts in order to have a global vision of the problem. We then start to make several prototypes and evaluate them rigorously. In Table 1 seven guidelines to be respected during the research project:

Table 1: Design Science Research Methodology [2]

Guideline	Description	Adaptation for this project
Design of an artifact	The research must produce a functional artifact	By artifacts we mean a prototype. In this project the artifact will be the interface for managing the enclave
Relevance	Research must solve a business problem by offering a technical solution	The problem is to be able to quickly manage the enclave without CURL requests
Evaluation of the artifacts produced	The utility, effectiveness and quality of an artifact must be rigorously demonstrated by evaluating it using appropriate methods	Once the interface has been developed, it must be tested by the members of the Softeng group to validate it. Then conduct various tests both in terms of software quality and business problem resolution.

Research contribution	The artifact must make a specific and verifiable contribution to the scientific field	In our case, our artifact would help the security field in IoT and automate the treatment of data emitted from Things
Rigorousness	Research must be rigorous, using methods of creation and evaluation derived from knowledge in the field	The application must be tested, and the results returned must be consistent and meet the expectations
Implementation of an iterative research process	The process of creating the artifact is iterative	The first solution is never the right one. Collaboration with Softeng group members is required to create a usable artifact
Communication	Research must be presented for both specialists and non-specialists	The defense of the master thesis must be accessible to a non-specialist audience

2.3 The Internet of Things (IoT)

To fully understand this project, it is first necessary to introduce the notions of the Internet of Things, its ins and outs. The Internet of Things is a network of objects connected to the internet that exchange information between them or to a server. We use the term IoT to refer to the Internet of Things. We will also refer IoT “object” as Things in this document.

These Things can be physical objects that take physical measurements such as atmospheric pressure and share this data with other Things or to a server. Things use the internet as a medium of communication. A Thing can also be dematerialized, such as software components that sends information on a regular basis [3].

More simply, Things, whether software or physical, collect data and then share it with other Things or servers through the internet. We can take as an example of a connected object a sensor that measures the rain level before sending it to a server for analysis.

The challenge is not to program these Things. Nowadays, it is very simple to program them with very little knowledge. There exist embedded systems with an impressive number of sensors that are very easy to program, even for a novice. Their price is also insignificant.

The great challenge nowadays is to be able to receive, store and process the data from these sensors. Extracting data in real time has become a major challenge. For example, if a sensor senses an earthquake, it is useful to be notified as soon as possible so that the population can be evacuated.

As objects all become connected, it has become extremely difficult to extract information, analyze it and exploit its potential. The challenge nowadays is to find solutions to exploit the incredible amount of data coming from sensors in order to trigger real-time actions.

The Internet of Things offers a totally unreasonable number of practical applications. The classical problem of integrating the different heterogeneous components thus arise. How to integrate Things with such different functions into the same information system? This is an issue that will be raised in this work. Simplicity and reusability must be emphasized in order to achieve full information system integration.

A second aspect of the Internet of Things is data security. Sensors can sense data that are potentially confidential. And the proliferation of this data that passes through the internet poses security problems. It is possible to either intercept data being transferred from one Thing to another or to decrypt it directly on the cloud virtualization platforms. Indeed, we consider cloud computing platforms unreliable and we cannot trust them. There exist secure execution environments available for running programs in hostile environments.

During this work, we will cover the challenges of data integration in the field of IoT and the security that goes with it.

2.4 Project Context

In the context of research in the Internet of Things, the Software Engineering Group of the University of Fribourg is developing a privacy-preserving middleware namely Intel Software Guard Extensions (SGX). This middleware is capable of interacting with smart devices (Things) and clients, while all the application data are hidden from the platform hypervisor, enabling trusted computing in untrusted environments. In particular in cloud computing.

At its current state, the system proposes a REST interface, using symmetric encryption to safely transmit HTTP (JSON) message with both Things and clients. The middleware receives sensors events, and depending on given dynamic rules, is able to trigger actions to either Things or clients. The middleware assumes the platform owner is completely untrusted [1].

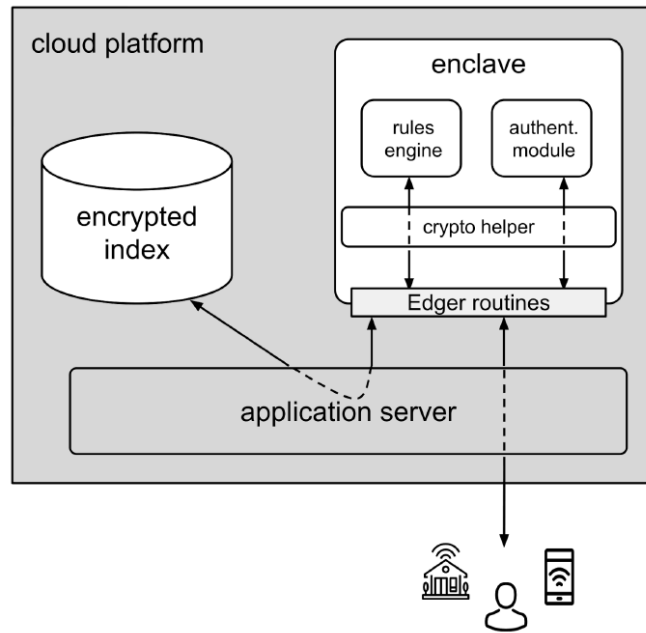


Figure 1: The secure middleware architecture. Illustration from P. Gremaud [1]

2.5 Project Objectives

The main goal of the thesis is to develop a client web interface to interact with the backend middleware. We will call this backend an « enclave ». Because the context of the project is data privacy in IoT, the interface should not store any data on the server hosting the interface [4], making it a « trusted » component of our framework. For this same reason, JavaScript is preferred as the main programming language. The Web Crypto API offers an implementation of most existing encryption standards and will provide cryptographic capabilities to the interface.

The interface should act as an administration tool able to:

- Authenticate the user with the middleware and generate a session key.
- Manage clients and Things.
- Manage events, actions and rules.

All communications exchanges with the middleware are done using encrypted messages packed in a JSON format.

The project goals can be listed as follows:

- Study and get familiar with the different technologies involved in the project, including TEEs and more precisely Intel SGX but also encryption standards.
- Study and get familiar with the existing middleware solution.

- Understand the encrypted communication protocol in order to communicate with the middleware.
- Define and prioritize a list of functionalities for the client interface. Depending on the available time, some of them may not be implemented.
- Find a suitable (JS) framework to develop the interface and get familiar with it.
- Implement the proposed interface. The developed solution should remain easy to modify and extend.
- Create a concrete scenario to showcase the interface by using existing, available things or by modifying one to suit the needs of the scenario.
- Write a thesis report describing the whole project.

The main goal of the project is to develop a graphical web interface to manage the enclave. However, as the enclave running on Intel SGX is not ready yet, it will be necessary to develop a second enclave simulating the one that will be developed in order to complete the project. The auxiliary components will also have to be developed. It will not be necessary to develop an enclave that is compatible with Intel SGX.

2.6 The iFlux project

2.6.1 Introduction

The iFlux project [5] is led by the HEIG-VD. It was created to provide a lightweight middleware for integrating Things and data in smart cities. It is based on three abstraction principles, which are *event sources*, *action targets* and *rules*. iFlux makes it very easy to expose a REST API to sensors and actuators. It provides to Things a very high-level API to integrate them into a heterogeneous ecosystem and put them into a workflow. Things can be intelligent objects or pure software components and iFlux makes no difference between them. The iFlux programming model allows to integrate data of these Things in order to be able to trigger actions based on event data with rules. iFlux focuses on smart cities but its scope can, of course, be extended to other domains.

We will use this project as a model to develop our own integration middleware.

In this section, we will introduce iFlux middleware and see how we can use it to create our own middleware that suits our needs.

2.6.2 Smart Cities

Information and communication technologies can improve the quality of life of a city's inhabitants by impacting a wide range of areas such as energy consumption, transport, security, public administration, politics or even culture [5]. Cities that use Things to sense data and use actuators are called smart cities. For example, we can imagine sensors under public parking places in order to instantly know what places are available. Another application could be to know the position of each person on the street in order to activate public street lights at night. The fields of applications are varied and unlimited.

Things that have a physical dimension will capture physical data and operate actuators. We can take the example of a gate that closes or opens automatically depending on the RFID chip that the wearer carries on him. Things can also be software only, such as the GPS position emitted by a smartphone and stored in a database. The synergy of physical components and software components makes a city smart. We make no distinction between the two types of Things because we use a very high-level programming model [5].

In this section, we will describe how the iFlux model works and how it allows cities to become smart.

2.6.3 The iFlux programming model

As we have seen before, Things can have a large number of practical applications. The integration of Things in an information system is a headache itself. In a "normal" information system, the integration of the various components is a challenge for developers. In the field of IoT this problem remains unchanged and may even be worse [5]. According to iFlux, the following questions arise:

- How to integrate heterogeneous systems scattered throughout the city?
- How to deploy Things and integrate them into the information system while guaranteeing a long-term support?
- How can we make the data generated by Things shareable and accessible?
- How to make the integration as simple as possible?

The iFlux programming model was inspired by the If This Then That (IFTTT) [6] programming model. It defines the paradigm events, conditions and actions (ECA). Three levels of abstraction have been defined based on ECA: the events source, the action targets and the rules [5]. Basically, it means that Things throw events, the events are caught, and actions are triggered according to some rules. A rule engine needs to be created in order to infer on data. Applications developers implement their workflow according to the ECA paradigm and actions are figured out by the rule engine [5].

The objective of iFlux is simplicity and reusability [5]. Developers can work on micro services and once these stand-alone components are completed, they can integrate them into iFlux. It is important to keep a minimum coupling between the different components because of the great heterogeneity of the different Things to integrate; having a strong coupling would be very difficult to achieve and maintain [6]. A weak coupling allows us to connect several autonomous components in an easy and permanent way. iFlux and all its components are based on the principle of micro services which allow a maximum decoupling. The communications are done with the JSON format, which is universal and easy to understand. All micro services in iFlux ecosystem must have a clear, documented and public REST API [5].

Event Sources

Events are data that are generated by Things that can either be hardware or software. A Thing emits a single event or flow of events. They can be regular or irregular. As examples of event sources, we can for example think of hardware Things that detect and send data on a regular basis, such as humidity sensors that send data on a regular basis. Software sensors can monitor stock markets and send alerts when a certain threshold is reached. Data processing services can merge data from several Things to form a new, more structured data flow. User agent Things can also transmit data very occasionally, for example, when there is an accident and an alert is issued [5].

The iFlux middleware exposes a public and documented REST API that allows Things to send data to the middleware. The API defines that an event is composed of data, a source client, an event type, and a list of allowed parameters. These parameters depend on the type of event.

The events endpoint only accepts POST requests because an event is unique and unchangeable. An event can contain a single set of values or an array of values. Code 1 shows that it is possible for any Thing to send data in a completely decoupled way by specifying an event type corresponding to the data sent in the event.

```
1  POST /events http/1.1
2  Content-type: application/json
3
4  [{
5      "timestamp": "2015-01-12T05:21:07Z",
6      "source": "/event-sources/JI8928JFK",
7      "type": "/eventTypes/temperatureEventSchema",
8      "properties": {
9          "temperature": 22.5,
10         "location": "room 1"
11     }
12 }]
```

Code 1: iFlux event

Action targets

The action target is the action that will be triggered by the middleware if the rule engine has identified a positive condition to a rule based on an event. When a Thing sends an event to the middleware, it will look for all the rules associated with that client and this event type, then evaluate all the rules. If a rule is evaluated positively, then the middleware will execute the action associated with that rule. The action target is a micro service and must expose a REST API that must be accessible by the middleware. This API will then be called to execute the action defined in this rule [5].

An example of an action would be to send an email to a recipient or to send a push notification. The action targets must be implemented by developers or third parties service providers.

Code 2 shows an example of an action target. As we can see, an action target is composed of an event type and a rule. If the rule is triggered, then the action in the properties are triggered on a certain URL. A POST request is then made, and the body request consists of the properties in Code 2.

```
13  POST /rules/myAction HTTP/1.1
14  Content-type: application/json
15
16  {
17      "context": {
18          "event": "events/gatherDataFromRoom",
19          "rule": "rules/alertWhenRoomIsTooHot",
20      },
21      "type": "/actionTypes/sendAlertViaEmailSchema",
22      "properties": {
23          "email": "user.name@iflux.io",
24          "subject": "Alert: something has happened!",
25          "body": "An event has been notified to iFLUX by a source and a
rule states that we should inform you about it."
26      }
27  }
```

Code 2: iFlux action target

Rules

The endpoint of the rules is used to specify the rules that will be tested in order to trigger or not an action. A rule is used to link an event to an action. When a Thing throws an event, the middleware will, thanks to its rule engine, infer on the rules in order to find an action that corresponds to an event. If the condition present in the rule is positively evaluated, then the associated action is executed. It is then possible to build rules allowing condition like: *if* the temperature of the chamber is higher than 25 degrees *then* a push notification is sent [5].

As we can see with Code 3, a rule is composed of an event type and a Thing source. If the condition is evaluated positively then a POST request is made with a certain URL with a mandatory list of fields.

```

28  POST /rules/mahRule HTTP/1.1
29  Content-type: application/json
30
31  {
32    "description": "Hot temp ? Notify Bob by email.",
33    "reference": "TEMPERATURE-EMAIL-NOTIFICATION",
34    "enabled": true,
35    "if": {
36      "eventSource" : "/event-sources/JI8928JFK",
37      "eventType" : "/eventTypes/temperatureEventSchema",
38      "eventProperties" : {}
39    }
40    "then" : {
41      "actionTarget" : "https://mail-gateway.iflux.io/api",
42      "actionSchema" : "{\"type\" : \"sendEmail\", \"properties\" :
43        {\"to\" : \"bob@iflux.io\", \"subject\" : \"New temperature\",
44        \"body\" : \"The temperature in is now: .\" } }"
45    }
46  }

```

Code 3: iFlux rules

Recap

As we have seen with the iFlux project, it is possible to integrate Things into the workflow of an information system without having to work intensively on software integration. iFlux operates on three high-level principles: events, rules and actions. A Thing throws an event to the middleware and the latter will infer on its rules thanks to its rule engine. Once a rule is positively evaluated then the associated action is executed. An action is a micro service that exposes a REST API and that can be called by the middleware. They allow Things to POST events following a certain protocol and it only remains for the developer to specify which actions to perform based on the data received by Things.

The components diagram in Figure 2 shows how iFLux works in a higher level. We can see a sensor or Things that sends an HTTP request to the middleware. iFlux API catches the request, check if it's valid and then infer on the data with its rule engine. If a rule is evaluated positively, then the middleware will trigger an action by making an HTTP request on an actuator that also exposes a REST API.

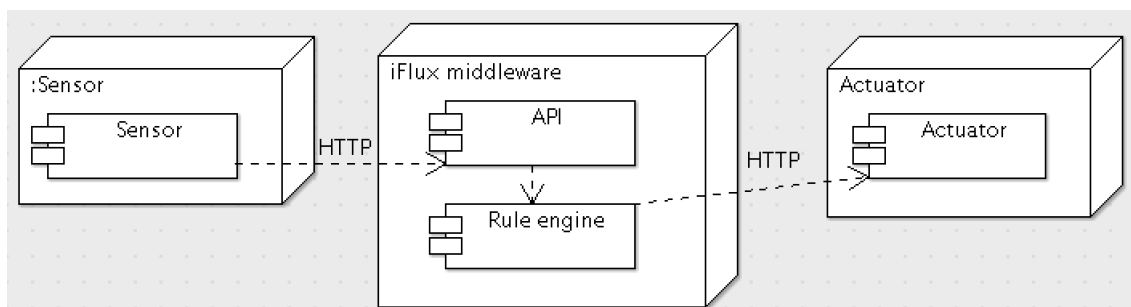


Figure 2: iFlux components representation

During this project, we will create our own version of iFlux based on its core principles. It includes the concepts of events, rules and action. We will modify it to include our safety requirements and some of our technical specifications. The practical part of the document covers the difference in detail.

2.7 Trusted Execution Environment (TEE)

2.7.1 Introduction

The overall goal of this project is to develop a web interface to manage a middleware that runs on a protected environment. Even if we are not going to develop software that will run on these protected environments, it is necessary to theoretically cover their existence in order to better understand the issues of the project and develop an application that will best serve our interests and needs.

Trusted Execution Environments (TEEs) are memory areas in the CPU that are isolated from the operating system. These areas ensure that the data in these presents are stored and processed in a totally secure way.

The main concepts of TEEs are trust, security and data isolation [7]. They provide to Things point-to-point security by identifying them and encrypting data. All data in the TEEs are encrypted as well as the applications running on them. Applications running in TEEs are called trusted applications [7]. Data handled by these trusted applications are encrypted and executed securely without any other application being able to intercept the data being processed. For example, if the architecture offers TEEs functionalities, it can directly access devices such as the camera or fingerprint reader in order to securely process this data [8]. Communications are encrypted between the components, which allows for instance components to communicate with the database in a totally secure way, for example. All entities communicating with an application running on a TEE must be authenticated and authorized [9, 10].

TEEs are important nowadays because we live in a society where data is everywhere, every time. We use digital tools to communicate by email or chat. We use the internet to order products. More and more common devices are connected to the internet. We produce a huge amount of data every day and it's becoming essential to protect this storm of data. Data protection is nowadays essential to prevent our data from being intercepted for misuse. TEEs in this context play a role in protecting our data against espionage by third parties. The internet needs trust and scalability. Being able to manage data securely and scale up is the key for offering a competitive service in the marketplace. TEEs implement data security by encrypting communications and data processing. Encryption is now assisted by hardware, so it is very powerful and can be scaled up very easily [10, 9].

TEEs provide new features to support security and scalability in hostile environments. They allow the development of new technologies as well as services that were previously considered difficult to provide. We will see in detail in the following part of this document a specific implementation of TEE which is Intel SGX and how it is used in our context.

2.7.2 Intel SGX

Intel offers its own implementation of TEEs called Intel Software Guarded Extension (SGX). The principle is that SGX allows developers to run their applications in protected memory areas called *enclaves* thanks to the different instruction sets implemented directly in the Intel CPUs.

Deploying an application on the cloud is now really common. The problem is that our application that will run on the hypervisor of the cloud can be compromised and data intercepted by the cloud provider itself. The environment in which our application is deployed is considered hostile and thus untrusted. If the cloud provider exposes the possibility of using Intel SGX, then we can create secure enclaves isolated from the indiscretions of other applications and even the cloud provider.

Intel SGX allows the creation of enclave applications composed of both untrusted and trusted parts. The goal is to use the trusted parts as little as possible, just enough to protect our secrets [8], and to remain as efficient as possible because calls in the trusted zone are expensive. Intel SGX does not replace good programming practices, which are to ensure that data is always consistent and that no buffer overflow is possible [1, 7, 8].

Figure 3 shows an application running on a cloud that offers Intel SGX capabilities. The untrusted part of the enclave normally runs when the trusted part is directly executed in the protected memory. When the untrusted part needs to do a protected action, it calls a method that sends data to the protected/trusted part. The trusted part sees the data in clear and returns the result when finished.

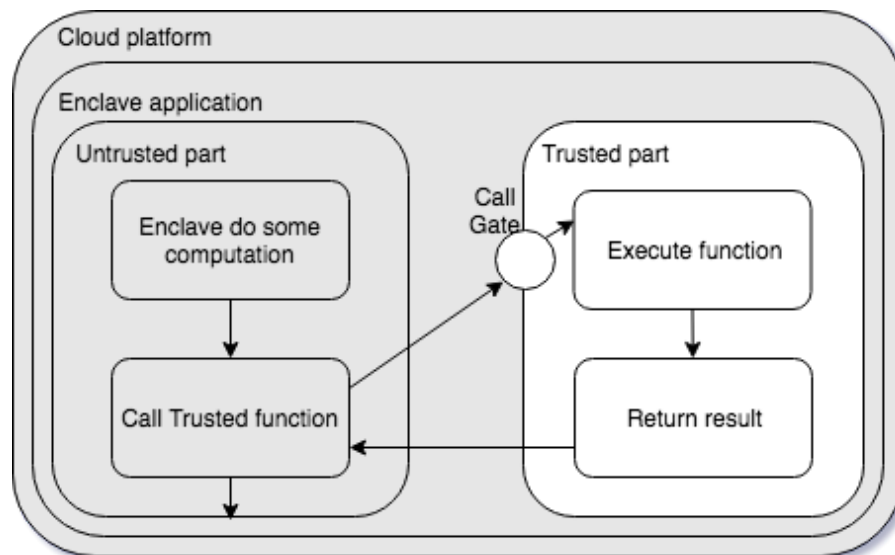


Figure 3: Intel SGX enclave, trusted/untrusted parts

2.8 The IoT middleware

The Software Engineering Group of the University of Fribourg is developing a secure Privacy-Preserving IoT middleware using the Intel SGX CPU set of instructions. The goal of the project is to propose an application that will expose a REST API to Things [1]. This API will then, thanks to a rules engine, orchestrate data coming from various Things, then execute actions according to rules. The iFlux programming model is the inspiration behind the whole process. This middleware is deployed in a cloud provider that is considered untrusted but offers the Intel SGX instructions set. The goal is therefore to develop an application running in an enclave, which exposes a REST API to process data from Things [1, 5, 8].

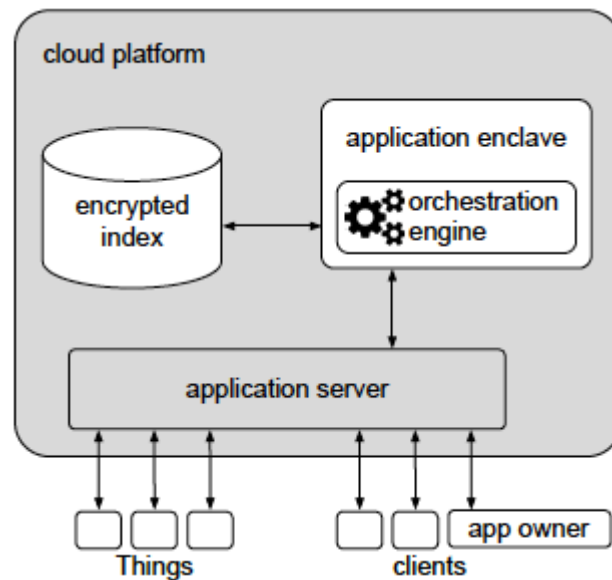


Figure 4: General architecture of the system. Illustration by P. Gremaud [1]

Figure 4 shows what is the middleware current development state. The gray parts are the untrusted part while the white parts are the trusted part. The cloud platform is considered untrusted and the enclave runs the rule engine. Data generated by Things as well as the source code of the enclave are not accessible by anyone other than the instance of the enclave being executed [1]. Data in the database do not run in an enclave but are considered trusted since they are stored as encrypted data.

Intel SGX was not designed to protect a large portion of data. That's why there is a database containing the different information needed to run the middleware. The data inside the database are encrypted by the enclave and only the enclave can decrypt it with its private key. As data are encrypted, we can consider the database to be trusted. The application server is a REST API that runs on the cloud to communicate with the rest of the world. It is considered untrusted because it does not run in a protected zone of the memory. As the application server runs in a cloud considered untrusted, it is necessary to encrypt all communications from Things as well as from clients. The REST API will therefore intercept the requests in an encrypted manner, then send the results to the enclave. The latter will decrypt the messages, then infer on the data received with its rule engine in order to trigger the necessary actions. Messages sent in action targets are also encrypted because they are considered to be sent in untrusted environments. There is a semantic distinction between clients and Things. In general, Things are sensors that

send data to the enclave. Clients are the owners of the enclave application who have the task of managing the enclave and its rule engine [1].

As we have seen, an enclave is used to execute code in a protected area of the memory. All the information contained in it is not visible to anyone else. However, there is a security issue during communications. All participants in the system exchange information with the HTTP protocol and pass data in clear text. This information may be intercepted by third parties or the cloud provider. This is why it's necessary to encrypt communications between Things, clients, the application server and the enclave. It is therefore necessary to have an encryption system and a protocol to respect that allows to secure the data that is transmitted between the different components. Things and clients must be able to exchange public keys with the server. Standard symmetric encryption algorithms are then used to encrypt all exchanged messages in order to secure the entire chain [1].

We saw in this section that a middleware based on the iFlux model is under development. This middleware runs in an enclave to ensure data security and confidentiality for users. However, this middleware is still under development and it will be necessary to redevelop our own middleware application for the needs of this project. For reasons of convenience and simplicity, the middleware developed will not be compatible with a TEE. The goal of this project is to develop a web client that manages the middleware data in order to run the rule engine. This client must also be able to manage data encryption. We will see in this report how we have developed our client and how we manage encryption with our middleware.

2.9 Communication protocol

2.9.1 Introduction

In the middleware currently under development, clients and Things exchange messages with the enclave through a REST API. The communications are therefore made using the HTTP(S) protocol. Encryption of messages is necessary because the HTTPS protocol allows the confidentiality of data from the client to the server. This is problematic in our case because our server is considered untrusted. It is then necessary to encrypt all communications to ensure that messages are transported securely up to the enclave. In this section, we will see which cryptographic techniques we will use as well as the protocols implemented to ensure connection establishment and symmetric encryption.

2.9.2 Key exchange algorithms

Diffie-Hellman (DH)

Symmetric encryption algorithms are very common and powerful. AES for example allows us to quickly encrypt a large amount of data. It is standardized and widespread. However, it is necessary to have an identical private key (shared secret) on each side in order to encrypt data. It is therefore necessary to have a key exchange system. The establishment of an encrypted connection takes place in two steps. The first step is to exchange both sides public key. Then the client and server agree on a private key that will be used temporarily to encrypt communications and that can be revoked at any time. This key is unique for each client. This process enables the exchange of encrypted messages with a symmetric algorithm such as AES. The latter will use the key that was defined by Diffie-Hellman [4, 1, 11, 12].

The Diffie-Hellman (DH) algorithm is widely used for key exchange. The purpose is to create a private key (shared secret) between a client and a server via the exchange of messages over an unsecured channel. We will see that even if the messages allowing the establishment of the key are sent in clear on an unsecured channel, it is difficult to deduce the private key for an attacker who intercepts the messages [11, 13].

The session key must not be stored permanently by the client. This ensures Perfect Forward Secrecy (PFS). If an attacker records each message and manages to get the private key, he could only decipher the messages that have been ciphered using the current session key but could not uncipher previous or future ones [4].

Let suppose we have Alice (the client) who wants to agree with Bob (the server) on a private key and Takeshi (the thief) listens to the communications between Alice and Bob. The steps for establishing a private key are in Table 2.

Table 2: Diffie-Hellman private key exchange steps

Steps	Alice	Bob
# 1	Alice and bob choose together a very large prime number p and a large number g such that $g: 1 \leq g \leq p - 1$.	
# 2	Alice chooses a very large random number x	Bob chooses a very large random number y
# 3	Alice computes: $p_1 = g^x \bmod p$	Bob computes: $p_2 = g^y \bmod p$
# 4	Alice and Bob share theirs values of p_x through an unsecure channel	
# 5	Alice computes the secret key $k_1 = p_2^x \bmod p$	Bob computes the secret key $k_2 = p_1^y \bmod p$

The example in Table 3 shows the establishment of a private key on both Alice and Bob with real numbers:

Table 3: Example of Diffie-Hellman with numbers

Steps	Alice	Bob
# 1	$a = 3, g = 17$	
# 2	$x = 15$	$y = 13$
# 3	$p_1 = 3^{15} \bmod 17 = 6$	$P_2 = 3^{13} \bmod 17 = 12$
# 4	$P_2 = 12$	$p_1 = 6$
# 5	$12^{15} \bmod 17 = 10$	$6^{13} \bmod 17 = 10$

As we can see, in step 5, the calculation ends with the same number, which gives the secret key. Even if the two equations don't look the same: $12^{15} \bmod 17 = 6^{13} \bmod 17$, they can be rewritten as $a^{yx} \bmod g = a^{xy} \bmod g$ or $3^{13 \cdot 15} \bmod 17 = 3^{15 \cdot 13} \bmod 17 = 10$ and we can see that they both do the same calculation with the same exponents but in different order and end up with the same result that will be used as the private key. Confidentiality is guaranteed by the fact that if T (thief) intercepts communications between Alice and Bob, T would have no reasonable way to find out the private key from the information transmitted through the unsecure channel. X and y being very large numbers, it is indeed extremely complex to find their value from the information transmitted in clear in a reasonable amount of time [11, 13].

Elliptic Curve Diffie-Hellman (ECDH)

The previous section presented the Diffie-Hellman algorithm. However, in this project, another version was used. It's the Elliptic Curve Diffie-Hellman. It was necessary to introduce DH first because it is easier to understand than ECDH. The main difference is that DH uses modular arithmetic to define a secret key while ECDH uses algebraic curves [14]. The maths behind ECDH it will not be explained here, but the principle of key exchange remains the same as with DH.

Shared Secret derivation with SHA-256

Once a secret key has been established with the ECDH algorithm, it is then necessary to derive the key. It is not advisable to use the secret key directly [15] because a secret key is not uniformly random, and an attacker could take advantage of partial information to deduce our secret key. This is why we will derive the keys with the SHA-256 algorithm to create a 256-bits hash and use it as a private key [13, 16].

2.9.3 Advanced Encryption Standard (AES)

To encrypt communications, we will use the Advanced Encryption Standard (AES) algorithm. It is a symmetric encryption algorithm that allows data to be encrypted in 128-bit blocks with a 256 bits private key in our case. The advantage of AES is that its specification is open source, it is very resistant to attacks, easily implemented by both software or hardware components and provides very good encryption performance [15, 17]. In this project, we will use the AES-GCM implementation (Galois/Counter mode).

AES operates on 128-bit blocks of plain text that it transforms into 128-bit encrypted blocks by a sequence of n rounds. A 256-bit key will require 14 rounds. The simplified Figure 5 (homemade) shows how the AES algorithm works for ciphering and enciphering text [18]:

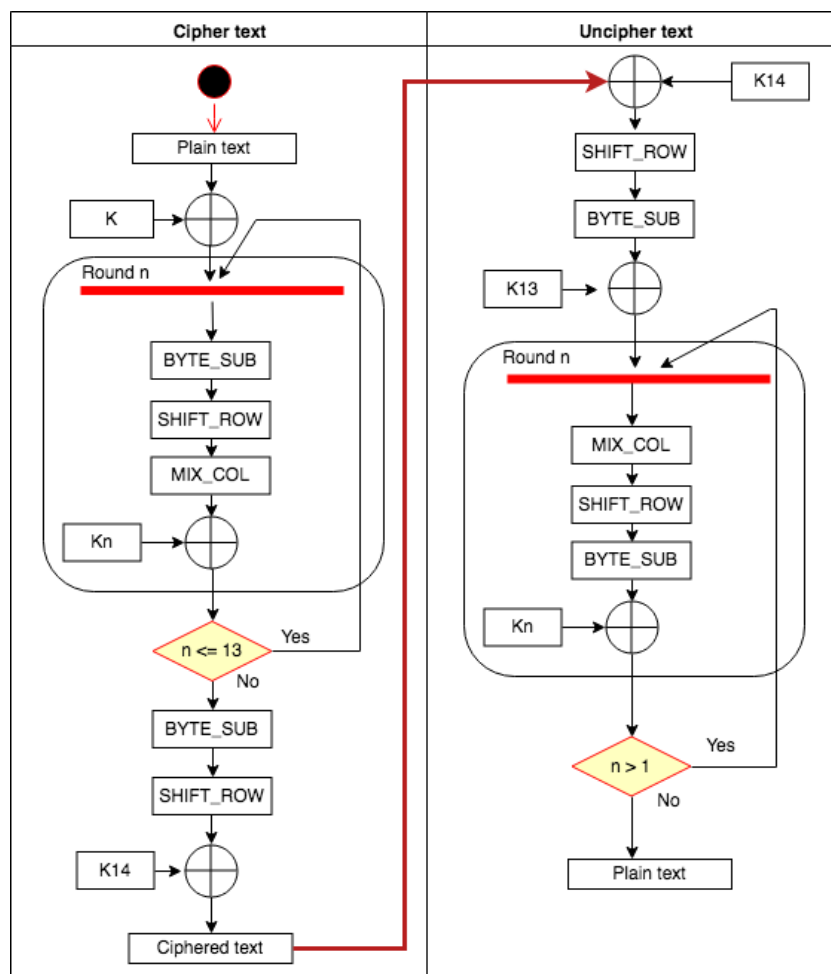


Figure 5: Graphical example of the AES algorithm

All the function descriptions were taken from D. McGrew and J. Viega paper on AES-GCM algorithm [15].

BYTE_SUB is a non-linear function that operates independently on each block from a so-called substitution table.

SHIFT_ROW is a function that operates offsets (typically it takes the input in 4 blocks of 4 bytes and operates offsets to the left of 0, 1, 2 and 3 bytes for pieces 1, 2, 3 and 4 respectively).

MIX_COL is a function that transforms each input byte into a linear combination of input bytes and can be expressed mathematically by a matrix product on the Galois.

K_n is the i th subkey calculated by an algorithm from the main key K .

Uncipher a message consists of applying reverse operations, in reverse order and with subkeys also in reverse order.

Another mechanism used with AES to increase security is the *iv*. An Initialization Vector is the initial value used to start some iterated process. It prevents repetition in data encryption to make harder to an attacker to use a dictionary to find patterns and break the cipher [19]. AES-GCM requires a 96 bits random *iv* [20] that is unique for each message encrypted for a given private key. A random and unique *iv* ensure that each message is ciphered differently. The IV can be kept public and must be in our case higher than the previous one.

2.9.4 Message exchange protocol

Introduction

As we have seen in the theoretical part, the enclave runs in an environment considered untrusted. Communications can be intercepted either during data transmission over the internet or directly on the cloud provider platform. Encrypting the data is then necessary and, in this section, we will explain the protocol used to guarantee the security of the data. We assume here that a client wants to exchange data with the enclave and we will show how keys are exchanged and common secret generation and messages sharing are done. Messages exchanged with the enclave are sent over HTTP(S). They are ciphered at the application level and then HTTP(S) is used to transport them. Application encryption allows us to keep control over the encryption mechanisms and ensure that the data arrives securely to the enclave [4]. The protocol uses the JSON notation with all data coded in base64Url (without padding).

Session Key establishment

The first step to be able to communicate with the enclave from the client is to establish a session key. We will use ECDH to create a shared secret between the server and the client and we will use this key to encrypt all data [1, 4]. The first problem is that the enclave only accepts encrypted connections, and we need an unsecured entry point to establish a session key. Public key exchange can only be done via an unsecured channel. In our case, the enclave exposes a single URL that allows for key exchange and is located at *POST:/keys/exchange*.

Code 4 shows how the client initiate the communication with the enclave by sending its public key through an unsecured channel and the response from the enclave.

```

45 POST /1.0/keys/exchange HTTP/1.1
46 Content-type: application/json
47
48 {
49     "publickey": "3059301306072A8648CE3D020106082A8648CE3D03010703420
004809CB1C845DF75A504C6B1AC03F33D139DA99EEEA3E140433983C4D61CCF1D42CF25A7
F296816ABB7B49AA644FAAF3E5FC19A632C66EA764CA3E18B71FAEBE68"
50 }
51
52 Response Status 200:
53
54 {
55     "header": "B64Url("{\"alg\":\"dir\", \"kid\": \"123456789\"})\"",
56     "ciphertext": "B64Url(Ciphered(\"HELLO YOU =D Love from the enclave ;-
57     \"iv\": \"ASNfZ4mrze8BI0Vp\",
58     \"publickey\":
\"3059301306072A8648CE3D020106082A8648CE3D03010703420004E560F4DD90249D8A3C
FE2C545791D1344D8870D486B481E270D2CBC7C420761EF7CC3F881602B8C5E941CFD9E5C
8B4C6AA238B852A2D662539100B2E112E16E5\",
59     \"tag\": \"EZGft08x2pwMz5mZBnLV6g\"
60 }

```

Code 4: Example of session creation

Once the key exchange request is received by the enclave, the enclave will create a private key for that client, store it and return some data to the client. The received data is a JSON document composed of several properties. For now, the three properties we are interested in are the *publickey*, *iv* and *ciphertext* properties. The client will use the public key of the enclave to create its own private key. He will then decipher the *ciphertext* text using *iv* and if the client can decipher the text *"HELLO YOU =D Love from the enclave ;-)"* then it means that he has managed to create a connection with the enclave and has managed to derive a usable key between the two parties.

JWE (JSON Web Encryption)

JWE (JSON Web Encryption) is a specification that standardize the way we represent an encrypted data structure with JSON [21]. JWE can be represented in two different ways: the first is as a JSON document and the second is a compact form that is as a token serialized in Base64 separated with dots. Code 5 shows a JWE as JWT compact serialization.

```
B64URL(Header) . B64URL(Key) . B64URL(IV) . B64URL(CipheredText) . B64URL(Tag)
```

Code 5: JWE as JWT compact serialization example

The header contains the information in Code 6.

```
{"alg": "dir", "kid": <session key ID>, "cid": <client ID>}
```

Code 6: JWE header

The property *alg* with the value *dir* means that the algorithm used for encryption is direct. It means that the token is directly encrypted by the application. The *kid* is the id of the key used for encryption. The *cid* is the id of the client that has initiated the session.

The key part is the key used to encrypt the data. In our case, this part will be empty, meaning that we have used our own private key for encryption and we do not provide the key in the token. Even if we don't specify the key, we still have to include the dot and the compact JWE would look like: *header.iv.ciphertext.tag*.

The *iv* is the Initialization Vector used to cipher the text. We include it in our token and it will be used to decipher the text. The *iv* is always higher than the previous one [4].

The *ciphertext* part is the message that is ciphered. It is encoded in Base64Url without padding.

The Authenticated Tag is the final element of a JWE structure. It is used to ensure the integrity of the ciphered text.

We have seen what the compact representation of a JWE looks like. The JWE JSON representation has the same properties except that it is represented in JSON format instead of a string with dots. All our request body payload will be encrypted using the JWE JSON representation. Therefore, all the payloads of the messages will look like the JSON document in Code 7, except the one used to initiate a session.

```
61  {
62    "header": {"alg": "dir", "kid": <session key id>},
63    "iv": BASE64URL-ENCODE(<iv>),
64    "ciphertext": BASE64URL-ENCODE(<ciphertext>),
65    "tag": BASE64URL-ENCODE(<tag>)
66  }
```

Code 7: JWE JSON example

JWT (JSON Web Token)

A JWT is a token exchanged between a client and one or more servers allowing authentication and authorization access to certain resources for a given client. In our case, a JWS is represented as a JWT and is composed of a header, a payload and a signature. All parts are JSON documents serialized in Base64Url and separated by dots [21]. Code 8 shows a representation of a JWS as JWT token:

```
B64Url(header).B64Url(payload).B64Url(signature)
```

Code 8: JWS as JWT example

The header is composed of:

```
67  {
68    "alg": "dir",
69    "cid": <id of the client>,
70    "kid": <id of the secret key for this session>
71  }
```

Code 9: JWS header

The header in Code 9 part is composed with only essential information. The *kid* is the identifier of the session key used. The *cid* is the id of the client and the *alg* here specifies that we use direct encryption [4]. The payload is composed of:

```
72  {
73    "cid": <id of the client>,
74    "mag": "<signature of the JWE>",
75    "admin": 0|1
76  }
```

Code 10: JWS payload

The *mag* is present here is intended to make a link between the JWT present in the header and the JWE present in the body. Its value corresponds to the value of the JWE tag [4]. The *admin* property allows you to determine whether or not the client has additional rights.

A JWT is a token that a server distributes to a client after authentication. It is used to identify the client and contains in its structure information such as the identifiers and accreditation. This token is readable and editable by the client itself. It is therefore necessary to put additional protection on to ensure that the client has not changed the content of the token. The signature part allows the server to verify the authenticity and source of the token using the HMAC SHA256 algorithm. It creates a signature of the header and payload using a private key. This way, we can check if a user has modified the token.

After a request, the enclave returns a token in every case. However, the token is different in two cases: when a client requests a key exchange to open a session and the other case is for all the other requests. Indeed, we need a token to check if a client has access to a resource. Once a session is created, the enclave returns a token without *cid*. Once a client has logged in, the enclave returns a token with the *cid* in the header specifying that they are connected. Once connected, the token also changes with each request because the *mag* must match the JWE *tag*.

At each request, the token must be sent in the Authorization header like in Code 11.

```
Authorization: Bearer xxxxx.yyyyy.zzzzz
```

Code 11: Authorization header

The enclave returns a new token at each request. The client is expected to use this new token.

2.10 Conclusion

We have seen in the theoretical part that the Softeng group of the University of Fribourg is currently developing a middleware to manage data from Things. This middleware implements the latest Intel SGX instruction sets to execute code in a protected memory portion. The purpose is to ensure the execution of a program processing sensitive IoT data in an untrusted environment. We also saw classic methods of key exchange as well as symmetric cryptography algorithms allowing a client to create a session with the enclave, and so, offer point-to-point data protection. Data are then encrypted starting from the client, then arriving on the cloud provider before being sent to the enclave where the data will be decrypted.

We have also seen that the iFlux project proposes a programming model based on the three principles of events, conditions and actions (ECA). An event sent by a Thing is then submitted to a rule engine that will decide to trigger an action. We will therefore use the iFlux project as a basis for developing our own version of this ECA model.

The goal of this project is to develop a web interface with the latest technologies available to manage the middleware. As the project of the University of Fribourg is not ready yet, we will have to develop our own middleware that will simulate all the requested functionalities. However, it will not be necessary to make one that is compatible with a protected execution environment.

The rest of the document will present the technologies that have been selected to develop the web application and the infrastructure.

3

Design and Implementation

3.1 Introduction	39
3.2 Risks analysis	39
3.3 Global architecture	41
3.4 Cryptography middleware	42
3.4.1 Introduction	42
3.4.2 Web crypto API.....	42
3.4.3 Java Security.....	44
3.4.4 Benchmark.....	44
3.4.5 Spring Zuul.....	47
3.5 Custom REST API	49
3.5.1 Introduction	49
3.5.2 Keys.....	49
3.5.3 Users	50
3.5.4 Clients.....	51
3.5.5 Auth	52
3.5.6 Client Urls	52
3.5.7 Event types	53
3.5.8 Action types.....	54
3.5.9 Rules	55
3.5.10 Event	57
3.5.11 Conclusion	58
3.6 Enclave middleware	59
3.6.1 Introduction	59
3.6.2 Spring boot	59
3.6.3 Authentication/Authorization	62
3.6.4 Errors handling & Logging	64
3.6.5 Database type & model	66

3.6.6 Database access	68
3.6.7 Rule engine	70
3.7 Action target	78
3.8 Front-end	80
3.8.1 Introduction	80
3.8.2 Use cases	82
3.8.3 Navigation diagram	83
3.9 Practical example of cryptographic functions	85
3.9.1 Introduction	85
3.9.2 Session Key creation	85
3.9.3 Authentication	90
3.9.4 Message exchange	94
3.10 Software development organization	95
3.10.1 Source code version control	95
3.10.2 Iterative development	95
3.10.3 Testing	96
3.10.4 Deployment	97
3.11 Conclusion	99
3.12 Improvements & future works	100

3.1 Introduction

In the section 2, we saw the theoretical foundations that allow us to understand the stakes of the project and guide us through the realization of our software artifacts. In this section, we will use our preliminary analysis to define a global software architecture. We will then describe the technologies used and the important points of the technical implementation of the project. This part will guide us through all the important technical pieces of the project, the technologies, the architecture and the design.

3.2 Risks Analysis

In a project, a risk analysis is always important because it allows risks to be identified and quantified. Once a risk has been identified, it is then necessary to assess its probability of occurrence and impact on the project. For each risk identified, it is necessary to propose an alternative if possible or to propose a solution to limit its impact. Knowing the risks allows us to prepare for the difficulties that may arise and to find solutions.

This master's work being an original research work and based on the latest technologies, is fundamentally at risk. Indeed, there is a large part of unknown and ignorance about the ability to implement a particular feature based on technical knowledge or technical limitations. Table 4 shows the list of identified risks, their probability of occurrence and possible solutions.

Table 4: List of risks

#	Likelihood	Consequences	Details
1	3	1	Risk: Incomprehension of the specifications. The student cannot define the needs of the project
			Solution: Seek help from teachers, assistants and other researchers
2	4	1	Risk: Impossible to develop all the functionalities in order to correspond 100% to the original project. For example, develop an Intel SGX-compatible enclave
			Solution: Do not use Intel SGX. Develop as much as possible in order to best match the project specifications
3	4	2	Risk: Failure to successfully implement the full key exchange protocol
			Solution: Develop a simpler one
4	3	4	Risk: Failure to understand the technical concept of cryptography and technical implementation
			Solution: Use high level libraries

5	2	4	Risk: Web Crypto API (in development) does not work properly
			Solution: Use pure JavaScript cryptography libraries
6	1	2	Risk: Cannot develop a front end with the latest technologies
			Solution: Learn others front-end framework as alternatives. At least one should work
7	2	2	Risk: Cannot design an architecture that meets the needs of the project
			Solution: Seek help from teachers, assistants and other researchers
8	2	2	Risk: Cannot create our own API version of the iFlux project
			Solution: Seek help from teachers, assistants and other researchers
9	1	4	Risk: Cannot develop a rule engine in order to infer on events data
			Solution: Try to find a solution harder. It should not be that hard
10	2	2	Risk: The architecture is complex and hard to deploy
			Solution: Dockerize everything
11	2	4	Risk: Not enough time to complete the project
			Solution: Work harder and ask for more time

Table 5 is the risk matrix associated with the identified risks.

Table 5: Risks matrix

		Consequences			
		Minor (1)	Moderate (2)	Major (3)	Critical (4)
Likelihood	Unlikely (1)		6		9
	Possible (2)		7, 8, 10		5, 11
	Likely (3)	1			4
	Certain (4)	2	3		

The Table 5 graphically shows how the inherent risks of the project are distributed. As we can see, the main risks associated with the project have been identified and ranked on a personal scale. They are mainly of two kinds. The first risk is the understanding of the issues, needs and technologies related to the development of the project. Conceptual problems can easily be solved by asking for help. Technological risks can also be solved either by using existing solutions or by using high-level libraries if really necessary. Blocking risks have not been identified for this project. If a risk arises, there is always a solution to find an alternative or try harder to solve it.

3.3 Global Architecture

Once again, the goal of the project is to create a web interface allowing the management of a middleware similar to the iFlux project that proposes an events, conditions and actions programming model [4, 1]. The middleware we have to manage is currently under development, so it is necessary to develop our own infrastructure from scratch. However, it will not be necessary to create a middleware compatible with an SGX enclave, which will facilitate the development of our project.

Figure 6 shows the overall infrastructure as well as the rest of the infrastructure.

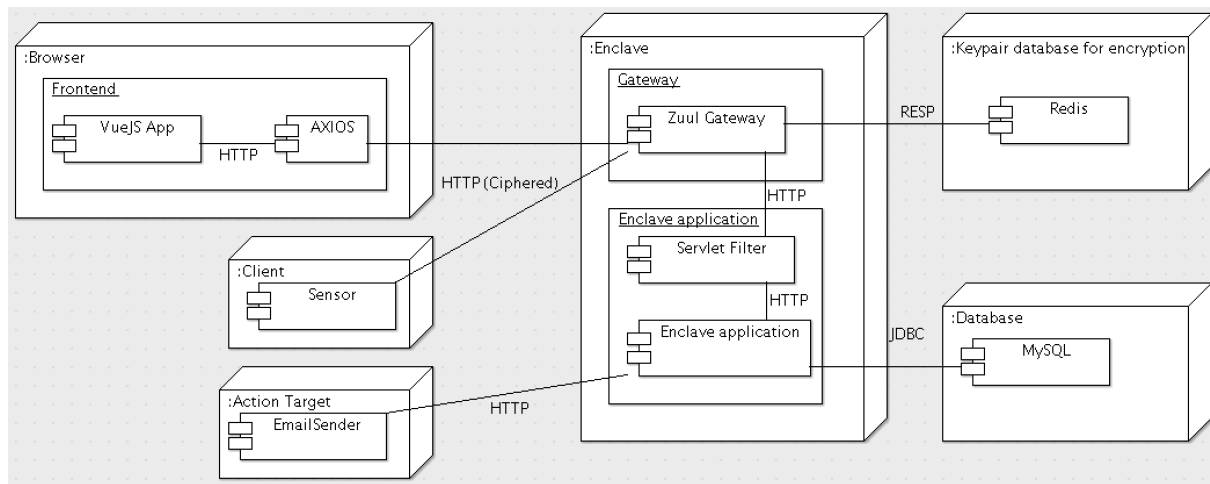


Figure 6: RIOT global architecture

The current architecture looks like the illustration in Figure 6. As we can see, it is a multi-tier architecture composed of several micro services. The front-end is a SPA (Single Page Application) that is used to manage the middleware. AXIOS is an HTTP client that allows us to make HTTP requests. Axios will then make requests to the Zuul gateway following the encryption protocol explained in the theoretical part. Zuul acts as a reverse proxy and intercepts requests, encrypt/decrypt them and send them to the servlet filter. The servlet filters will check the validity of the JWT tokens, then if it's valid it will send the request further to the enclave application.

We will call our middleware an enclave, even if technically it is not an enclave. Once the request is received by our enclave, it will then infer on it with the rule engine in order to trigger an action based on the events received. If a rule is evaluated positive, then the enclave will trigger an HTTP request to an action target. The action targets are developed with NodeJS.

The session keys generated are stored in a Redis database and all the data required to manage the rule engine are stored in the MySQL database. All data arriving at the enclave are decrypted, and it is Zuul's role as an encryption/decryption middleware to encrypt/decrypt data in order to make the enclave agnostic of any encryption mechanism. Things and clients are responsible for encrypting their own data before sending it to Zuul. Our web application is agnostic of all encryption mechanisms. Indeed, the interface makes a normal HTTP request. Axios will then transparently intercept the data, then encrypt it according to the protocol. The advantage of

making the front-end and enclave agnostic of encryption and thus offering a weak coupling between data and encryption is that if we want to get rid of any encryption or host our enclave on a trusted provider, it is then very easy to do so because it will not be necessary to remove all the encryption part in the code.

3.4 Cryptography middleware

3.4.1 Introduction

As we have seen before, the web interface must be able to manage the enclave in a secured way. The enclave is executed on an untrusted platform and all incoming messages must be secured. A session must be established, which means that public keys exchange is required. Then the exchanged messages must be encrypted, which means that another encryption algorithm must be used. Our web user interface is considered as a client with administrator privileges by the enclave. It is now time to ask the question of the technologies to be used to implement these cryptographic mechanisms. We have seen before which algorithms are used to exchange keys and to encrypt/decrypt. In this section, we will look at the technologies that will be used to implement encryption.

As a reminder, encryption takes place between the client and the enclave. However, as in this project, the enclave is not executed in a TEE, we will use other technologies that will help us to encrypt/decrypt our messages. The advantage is that the time required to develop an enclave is shorter as it's not the main goal of this project. On the web interface side, the data are transmitted by the Axios HTTP client. The data are then received by the Zuul reverse proxy before being sent to the enclave.

When the web interface makes an HTTP request to the enclave, the request is intercepted by Axios, then ciphered and sent to Zuul. The latter will take care of deciphering the message and send it to the enclave. Why did you intercept the messages with Axios and use Zuul as a reverse proxy, you may wonder? Simply to make the web interface and the enclave agnostic of any encryption. Indeed, as a web developer, it is not necessary to know the encryption details because they focus on feature developments. If someday day we decide to run the enclave on a trusted platform and encryption is no longer needed, then the weak coupling between the different layers allows us to remove the encryption layer very easily.

3.4.2 Web Crypto API

The web client must perform encryption. However, the scripting language used to execute code is JavaScript and until recently it did not contain any native cryptographic functions. In addition, JavaScript is considered slow and performing cryptographic functions with would be cumbersome.

Since 2016 the W3C has been proposing recommendations for a low-level interface (API) of cryptographic primitives [22]. This interface is implemented directly by the browsers. Calls to these implementations are made through calls of JavaScript function that return a promise. Once the computing is completed by the interfaces, the promises are then executed, and the results returned. Thanks to this API, we now have efficient functionalities allowing us to do hashing, key exchange and encryption. It should be noted that Web Crypto API does not offer a primitive to handle very large numbers. It is therefore necessary to use an external JavaScript library to manage the BigInteger, as it's needed for the *iv*. We should use Web Crypto API with caution as it is still in the experimental phase and is not fully supported by all browsers [22].

We have seen that Web Crypto Api offers an interface to use low-level primitives. However, this API is still under development and the implementation is partial in some browsers. There are many native JavaScript frameworks that existed long before. For example, we can mention SJCL, asmcrypto.js or CryptoJS [23]. These implementations have been in existence for several years and have been successfully tested. We can wonder why it would be necessary to use a native API when JS implementations already exist. The most logical answer is that JavaScript is slow, and a native implementation would be much faster.

The following benchmark was performed by the developers of the WebKit [23] engine that compares the native implementation with non-native implementations. The following algorithms were used:

- AES-GCM to encrypt a 4MB file.
- SHA-2 to compute the hash of a 512KB file.
- RSA to sign a verify the signature of a 512KB file.

The Figure 7 shows the result after multiple runs with the average value.

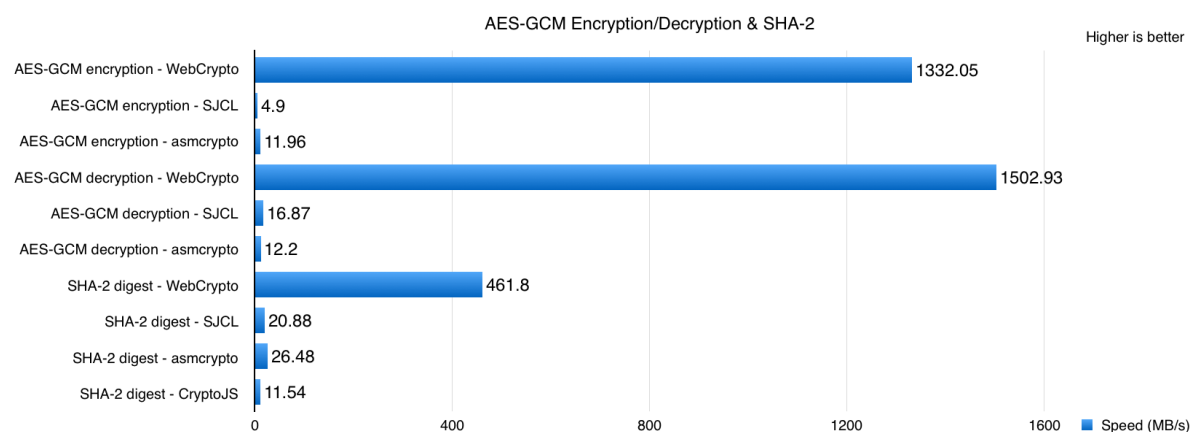


Figure 7: Web cryptography benchmark. Native vs JS implementation (Image from WebKit)¹ [23]

As Figure 7 shows, a native implementation of cryptography functions is much faster than JavaScript implementations. Web Crypto API is a major advance in cryptography for web applications. However, it is necessary to keep in mind that it is under development and is not fully supported [23, 22, 24]. The other issue is that there are very few examples of how to use the API and the documentation lack of clarity at the moment.

¹ <https://webkit.org/blog/7790/update-on-web-cryptography>

For the needs of our project, and as we use all the latest technologies, Web Crypto API will be privileged for its performance and by the fact that it integrates all the cryptography functions we need. Namely ECDH, SHA-256 and AES-256-GCM. BigInteger support is not available, we will then use the BigInteger.js library.

3.4.3 Java Security

On the server side, it is also necessary to encrypt/decrypt messages. Cryptographic operations require a large amount of resources, so it is necessary to use a compiled language rather than an interpreted language. NodeJS, even if it is very popular, would be a rather bad choice. We therefore need a language that provides cryptography primitives, that is fast and offers features to develop a REST API. The language retained is Java because it includes by default the *java.security* package which contains all the cryptographic functions we need [25]. It is also well known by the author of this document. The advantage of Java is that it is very widespread, easy to use and has extensive documentation on its cryptography part. It is stable and supported on the long term.

The *java.security* package contains all the necessary classes and interfaces to manage all the most common aspects of security and cryptography. However, the *java.security* package does not implement the algorithms by itself. It acts as a wrapper that provides a generic interface to a third-party provider [25]. The default implementation of cryptography algorithms is done by Sun Microsystems (legacy implementation) and other implementations can be specified by other providers such as OpenSSL for example.

Java security has a deliberate built-in limitation for a maximum key size of 128 bits. Indeed, if we try to use a 256 bits keys, Java will throw the exception *maximum key length permitted by policy*. It makes impossible to use the AES-256-GCM algorithm. Some people say and including this website [19] that this limitation is here because some countries have restriction on the permitted key size [19]. This limitation is easily overwritten by replacing files in *lib/security* of the JRE folder by the *Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy files 6* that can be found on Oracle's website [25, 19].

3.4.4 Benchmark

We have seen that Web Crypto API allows access to cryptography primitives implemented by browsers. This offers much better performance than pure JavaScript implementations. We also saw that Java provides a generic interface to call cryptographic primitives that are implemented by third parties. It would therefore be interesting to compare the performance of the two solutions to see if cryptography in the web environment is comparable to precompiled code.

In order to compare the performances, we need to do similar operations on both platforms even though they are not similar at all. Table 6 show the two scenarios that our platforms will compute.

Table 6: Java vs Web Crypto API scenario

Java scenario	Web Crypto API scenario
<p>Java is rather simple to monitor since the code is synchronous by design. We only need to measure the time elapsed from the beginning to the end.</p> <ol style="list-style-type: none"> 1. Create ECDH instance for both Alice and Bob 2. Print to the console their keypair 3. Create AES instance for both Alice and Bob 4. Exchange public keys 5. Creating a shared secret and derive a key from it 6. Generate the iv 7. Start the process of exchanging messages. This process will be executed N times: <ol style="list-style-type: none"> i. Defining a string message ii. Alice crypts the message with her derived key iii. The message is parsed to base64 iv. The message is parsed to string v. The message is decrypted by Bob vi. Increment the IV <p>The time elapsed is measured in seconds from before step 1 to after the last step of 7.</p>	<p>As JavaScript is asynchronous and the Web Crypto API massively rely on Promises, it's therefore much harder to measure performances as things don't go in a deterministic order.</p> <p>The following use case describe all the steps with the use of Promise and await/async.</p> <ol style="list-style-type: none"> 1. Create the ECDH instance for Alice 2. Extract public key 3. Create the ECDH instance for Bob 4. Extract public key 5. Derive a key for Alice 6. Derive a key for Bob with Bob 7. Start the process of exchanging messages. This process will be executed N times: Since this process is asynchronous, we'll rely on await/async mechanism to pretend we're in a synchronous mode <ol style="list-style-type: none"> i. Defining a string messages ii. Generate the iv (new for each exchange) iii. Crypt message with Alice derived key iv. Decrypt message with Bob derived key <p>As you can see here, for the sake of simplicity we didn't increment the iv and we skipped the parse to Base64 process.</p>

As we can see in Table 6, we have two similar scenarios that will each test the performance of cryptographic algorithms that have been implemented in Java and JavaScript. They will be executed 1'000'000 times and the average of the results obtained will be considered.

The tests were executed 1'000'000 times on Intel Core i5 2.7Ghz CPU with 4GB of RAM and in a single Java thread or browser tab. They both use ECDH with 256 bits key, AES-GCM-256 and 96 bits *iv*. The message that has to be ciphered/deciphered is “Hello Mah BOOOYYYYY! This is the raw message !!!” and weights 48 bytes.

Here is the result for Java:

```
Start benchmarking. Please wait...

Alice Private Key HEX      :
3041020100301306072A8648CE3D020106082A8648CE3D03010704273025020101042044B3F3826F19
A948080F1B0F5E59B89A7D38CFBC6B9A7696F21E4898F4F26CCB
Alice Private Key Base64 :
MEECAQAwEwYHkoZiZj0CAQYIKoZiZj0DAQcEJzAlAgEBBCBEs/OCbxmpSagPGw9eWbiafTjPvGuadpbyHk
iY9PJsyw==
Alice public Key HEX      :
3059301306072A8648CE3D020106082A8648CE3D03010703420004809CB1C845DF75A504C6B1AC03F3
3D139DA99EEEA3E140433983C4D61CCF1D42CF25A7F296816ABB7B49AA644FAAF3E5FC19A632C66EA7
64CA3E18B71FAEBE68
Alice public Key Base64 :
MFkwEwYHkoZiZj0CAQYIKoZiZj0DAQcDQgAEgJyxyEXfdaUExrGsA/M9E52pnu6j4UBDOYPE1hzPHULPJaf
yloFqu3tJqmRPqvPl/BmmMsZup2TKPhi3H66+aA==
Alice shared secret HEX :
25265F7A8CD55EECD1DFD7BDDA26662F4F9702E512A989ACAA75C7E69B9544BB

Bob Private Key HEX      :
3041020100301306072A8648CE3D020106082A8648CE3D030107042730250201010420320D1236B245
9D75FF3093DAA9675595B42FBE18AB7365C965179DF7BB994D56
Bob Private Key Base64 :
MEECAQAwEwYHkoZiZj0CAQYIKoZiZj0DAQcEJzAlAgEBBCAyDRI2skWddf8wk9qpZ1WVtC++GKtzZc1lF5
33u5lNVg==
Bob public Key HEX      :
3059301306072A8648CE3D020106082A8648CE3D03010703420004E560F4DD90249D8A3CFE2C545791
D1344D8870D486B481E270D2CBC7C420761EF7CC3F881602B8C5E941CFD9E5C8B4C6AA238B852A2D66
2539100B2E112E16E5
Bob public Key Base64 :
MFkwEwYHkoZiZj0CAQYIKoZiZj0DAQcDQgAE5WD03ZAKnYo8/ixUV5HRNE2IcNSGtIHicNLLx8Qgdh73zD
+IFgK4xelBz9nlyLTGqiOLhSotZiU5EAsuES4W5Q==
Bob shared secret HEX :
25265F7A8CD55EECD1DFD7BDDA26662F4F9702E512A989ACAA75C7E69B9544BB

Mode verbose deactivated

Start message exchange... Done
Benchmark Finished:

The whole process took: 11.394719722 seconds
There were 1000000 iterations
Each iteration took in average 1.3394719722E-5 seconds
```

Here is the result for JavaScript:

```
Start Benchmarking. Please wait... 1000000 iterations have to run...

Alice key pair generation done. Not extractable.
```

```

Alice's public key: {"crv":"P-
256","ext":true,"key_ops":[],"kty":"EC","x":"SylqEc9eSva0jWYxh56P7Ko4EYFNf0jkP3_e0
ZLahgc","y":"s6sd_uDLrz2p0hyD1-1AgEbM2tuhc8GDfv71FQhjIOs"}
Bob key pair generation done. Not extractable.
Bob's public key: {"crv":"P-
256","ext":true,"key_ops":[],"kty":"EC","x":"GkUMYeNs6WyTTPw9ySFuTBkPtHu3eyNPXpHiU
vANPfs","y":"YU9hhXfrqtopKHfwHXvFWln0XwzhORHIL7gnSjv67Q0"}
Alice derived shared secret generation done. Not extractable.
Bob derived shared secret generation done. Not extractable.

Verbose mode deactivated
Start message exchange... Done
Benchmark Finished:

The whole process took: 239.148 seconds
There were: 1000000 iterations
Each iteration took in average 0.000239148 seconds

```

We can obviously see which one is the fastest here. Java took 11 seconds to complete the whole process while JavaScript took 240 seconds. There is a 21-time speed up factor in favor of Java in this case. In other words, Java can cipher 91'000 messages per seconds or 4363 Kbytes per seconds. JavaScript can cipher 4'100 messages per seconds or 200 Kbytes per seconds. We now wonder why the JavaScript scenario is slower than the Java one. We know that Web Crypto API provides an interface to a native implementation provided by the browser. This means that the code called for cryptographic operations are compiled and should therefore be at least as fast as the code in Java, whereas this is not the case. The simplest explanation is that when JavaScript uses the API, it calls a function that returns a promise. Once the result is available, the API returns the result to the promise. Promises being pure JavaScript implementation, they provide much lower performance. This point may partially explain the relative slowness of Web Crypto API.

As we have seen, our Java implementation is unsurprisingly faster than its contender. It is therefore recommended to use Java as a server-side encryption language to take advantage of its performance because it may serve a larger number of concurrent users. As it will use several threads with Spring Boot, it will be even faster in the final implementation. Even if Web Crypto API is slower, it is fast enough to encrypt a smaller amount of data on the client side.

3.4.5 Spring Zuul

We have seen earlier that the front-end makes HTTP requests in a totally transparent way without worrying about encryption. The request is then automatically intercepted by Axios, then encrypted and sent to the enclave. We consider that the enclave must also be agnostic of any encryption. It is therefore necessary to have a mechanism similar to Axios that allows incoming requests to be intercepted, decrypted and sent to the enclave in clear text. Outgoing requests are also intercepted and encrypted. It is also necessary to manage private keys.

We must therefore provide a service that makes encryption transparent. This is where Netflix Zuul enters. Zuul is a project developed by Netflix and it can be seen as reserve proxy. Zuul was designed to be put in front of web services in order to add dynamic routing, monitoring, resilience, and security capabilities [26]. Zuul acts in a totally transparent way and in our case, it will be useful to encrypt/decrypt HTTP requests. Zuul was written in Java and can be used with services written in any language.

In this project, we will use Spring Zuul which offers a good implementation of Netflix Zuul. It will be placed in front of our API and it will intercept all requests. When a request arrives, the first thing to do is to check if it is a normal request or if it is a session creation request. If the `POST:/keys/exchange` URL is called, then the request is considered as a session request. Zuul will then send the request to the enclave and it will create a private key, store it in the Redis database, and then send the newly generated public key back to the client. Zuul checks that there are not more than 1000 session creation requests per IP for a period of 1 hour to avoid buffer overflow attacks. Once a session has been established, Zuul will then decrypt all requests, before sending them to the enclave. Requests returned by the enclave are encrypted before being returned to the client. Sessions keys are stored in the Redis database for a period of 1 hour and are indexed by *kid*.

We have seen in this section that it is possible to implement an encrypted and transparent means of communication for the API as long as the platform on which these two components are executed is considered trusted. The structure of the gateway source code is in Figure 8.

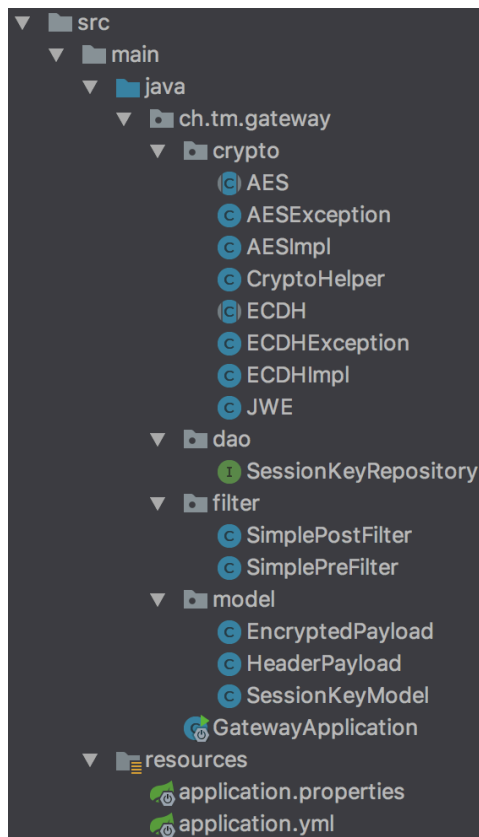


Figure 8: Gateway source code

The *GatewayApplication.java* file is the file that will execute the entire gateway.

The *application.properties* and *yml* files contain the gateway configuration variables.

The *crypto* package contains all the cryptographic functions.

The *dao* package allows access to the keys stored in Redis.

The *filter* package contains the code that allows messages to be intercepted, ciphered and deciphered.

The package *model* contains the session key templates in Redis.

3.5 Custom REST API

3.5.1 Introduction

We have seen that the iFlux project proposes a programming model based on the principle of events, conditions and actions. When an event occurs, it is evaluated, and an action is taken and triggered according to a condition. Based on iFlux's research, we have implemented a similar programming model. It is implemented as a REST API and will be detailed in this section. A Swagger documentation has been generated by the annotations that can be found in the enclave's code. We will briefly describe the different components of the API. For complete documentation, it will be necessary to run the enclave and go to the Swagger documentation.

A route version management has been implemented at the URL level. By default, all URLs have the version *1.0*. All HTTP verbs have been used and HTTP error codes and their messages are documented. The API described here is without any encryption.

3.5.2 Keys

Keys are used to manage sessions. When a client wants to communicate with the enclave, they must log in by sharing their public key. Code 12 shows how it's done.

```

77  POST /1.0/keys/exchange HTTP/1.1
78  Content-type: application/json
79
80  {
81    "publickey":
      "3059301306072A8648CE3D020106082A8648CE3D03010703420
      004809CB1C845DF75A504C6B1AC03F33D139DA99EEEA3E140433983C4D61CCF1D
      42CF25A7F296816ABB7B49AA644FAAF3E5FC19A632C66EA764CA3E18B71FAEBE6
      8"
82  }
83
84  Response status 201:
85
86  {
87    "header": {"kid": B64Url(kid), "alg": "dir"},
88    "ciphertext": "Ciphered(B64Url(HELLO YOU =D Love from the
enclave ;-)))",
89    "iv": "B64Url(iv)"
90    "publickey": "B64Url(enclavePublicKey)",
91    "tag": "B64Url(tag)"
92  }
```

Code 12: Session creation

As we can see here, in order to create a session, the client has to POST its public key. The enclave generates its private session key, store it in Redis and returns its public key to the client.

GET	/1.0/keys	keys
POST	/1.0/keys/exchange	exchangePublicKey
DELETE	/1.0/keys/{kid}	deleteKey
GET	/1.0/keys/{kid}	key

Figure 9: Keys API illustration

3.5.3 Users

As we said earlier, there is a distinction between users and clients. A client is a Thing that communicates with the enclave while a user is identical to a client except that it is specially created to use the web interface. A user is used to connect to the web interface to manage the enclave, sessions and other users.

```

93  POST /1.0/users HTTP/1.1
94  Content-type: application/json
95  Authorization: Bearer JWTWithAdminRights
96
97  {
98    "email": "marcel.grosjean@unine.ch",
99    "firstname": "Marcel",
100    "lastname": "Grosjean",
101    "password": "password"
102  }
103
104  Response status 201:
105
106  {
107    "insertedId": 1
108  }
```

Code 13: User creation

GET	/1.0/users	allUsers
POST	/1.0/users	createUser
POST	/1.0/users/hello	sayHello
DELETE	/1.0/users/{id}	deleteById
GET	/1.0/users/{id}	userById
PUT	/1.0/users/{id}	updateInfosById
PUT	/1.0/users/{id}/password	updateUserPassword
PUT	/1.0/users/{id}/roles	updateById

Figure 10: Users API illustration

3.5.4 Clients

A client is used to communicate with the enclave. It can perform all available actions if it has the admin role. If not, it will only be able to create an event. A client may have some URLs associated with it.

```

109  POST /1.0/clients HTTP/1.1
110  Content-type: application/json
111  Authorization: Bearer JWTWithAdminRights
112
113  {
114    "active": true,
115    "admin": true,
116    "name": "KitchenHumiditySensor",
117    "pubkey": "3059301306072A8648CE3D020106082A8648CE3D03010703420
004809CB1C845DF75A504C6B1AC03F33D139DA99EEEA3E140433983C4D61CCF1D
42CF25A7F296816ABB7B49AA644FAAF3E5FC19A632C66EA764CA3E18B71FAEBE6
8"
118  }
119
120  Response status 201:
121
122  {
123    "insertedId": 2
124  }

```

Code 14: Client creation

A client is identified by its name and we can pre-store its public key if needed. If the client is admin, it has access to the whole functionalities otherwise it can only post event. A client can be deactivated.

GET	/1.0/clients	findAll
POST	/1.0/clients	createClient
DELETE	/1.0/clients/{id}	deleteById
GET	/1.0/clients/{id}	findById
PUT	/1.0/clients/{id}	updateClient
GET	/1.0/clients/{id}/urls	findUrlsById

Figure 11: Clients API illustration

3.5.5 Auth

Once a session has been established, a key is stored on the server side. A JWT is returned to the client, but it does not contain any *cid*. The authentication step allows to obtain a JWT with a *cid*.

```
125 POST /1.0/clients HTTP/1.1
126 Content-type: application/json
127
128 {
129     "email": "string",
130     "password": "string",
131     "kid": "string",
132 }
133
134 Response status 200:
135 Authorization: Bearer JWTReturnedInHeaderBewareCORS
136
137 {
138     "token": "JWTToken"
139 }
```

Code 15: User authentication

The JWT is returned with the *cid* if errors 401, 403 and 404 have not been thrown.

POST /1.0/auth

createAllowedKey

Figure 12: Auth API illustration

3.5.6 Client Urls

A URL is associated with a client. When the rule engine has found an action to trigger then it will call the URL associated. The URL is the target of a micro-service that the enclave has access.

```
140 POST /1.0/urls HTTP/1.1
141 Content-type: application/json
142 Authorization: Bearer JWTWithAdminRights
143
144 {
145     "clientid": 1,
146     "value": "http://actuator01.unifr.ch/emails/tempalert"
147 }
148
149 Response status 201:
150
151 {
152     "insertedId": 3
153 }
```

Code 16: Url creation

GET	/1.0/urls	findAllUrl
POST	/1.0/urls	createUrl
DELETE	/1.0/urls/{id}	deleteUrl
GET	/1.0/urls/{id}	findById
PUT	/1.0/urls/{id}	updateUrl

Figure 13: Urls API illustration

3.5.7 Event types

An event type defines the format of an event that a client can post. The client that sends an event must specify an event type and must strictly comply with it. An event type is a list of allowed fields, their type (integer, number, string) and whether or not they are required.

```

154 POST /1.0/eventtypes HTTP/1.1
155 Content-type: application/json
156 Authorization: Bearer JWTWithAdminRights
157
158 {
159   "name": "TempEventType",
160   "scheme": {
161     "properties": {
162       "room": {
163         "type": "string"
164       },
165       "temperature": {
166         "type": "number"
167       }
168     },
169     "required": [
170       "room", "temperature"
171     ],
172     "type": "object"
173   }
174 }
175
176 Response status 201:
177 {
178   "insertedId": 4
179 }
180

```

Code 17: Event type creation

As we have seen here, an event type specifies which fields a client must provide when posting an event. In our case, a *TempEventType* event must have two mandatory fields that are the name of the room and its temperature.

GET	/1.0/eventtypes	findAll
POST	/1.0/eventtypes	createEventType
DELETE	/1.0/eventtypes/{id}	deleteEventType
GET	/1.0/eventtypes/{id}	findById
PUT	/1.0/eventtypes/{id}	updateEventType

Figure 14: Event type API illustration

3.5.8 Action types

An action type defines the format of the content body of the HTTP request that the enclave has to make when it wants to trigger an action. The rule engine, once a rule has been evaluated positive, will, depending on the data in the event, transform the data and send it to the URL (action target) in accordance with the action type.

```

181 POST /1.0/actiontypes HTTP/1.1
182 Content-type: application/json
183 Authorization: Bearer JWTWithAdminRights
184
185 {
186   "name": "TempActionType",
187   "scheme": {
188     "properties": {
189       "from": { "type": "string" },
190       "to": { "type": "string" },
191       "subject": { "type": "string" },
192       "content": { "type": "string" },
193       "temperature": { "type": "number" },
194       "room": { "type": "string" }
195     },
196     "required": [
197       "from", "to", "subject", "content", "temperature", "room"
198     ],
199     "type": "object"
200   }
201 }
202
203 Response status 201:
204
205 {
206   "insertedId": 5
207 }
```

Code 18: Action type creation

As we have seen, the action type defines the format of the HTTP request that the enclave will have to make to the action target. In our case, the request must have the following mandatory properties in its request body: *from*, *to*, *subject*, *content*, *room* and *temperature*. These properties are necessary to build and send an email. In the *TempEventType* event type,

we had the room and temperature properties only. As seen, we can add properties that don't exist in the events and we could imagine a templating system that would put in the *content* property the following values *"Alert the room {{room}} has a dangerous temperature of {{temperature}} degrees celcius."*

GET	/1.0/actiontypes	findAll
POST	/1.0/actiontypes	createActionType
DELETE	/1.0/actiontypes/{id}	deleteEventType
GET	/1.0/actiontypes/{id}	findById
PUT	/1.0/actiontypes/{id}	updateActionType

Figure 15: Action type API illustration

3.5.9 Rules

Rules are the central concept of the rule engine. They allow the enclave to trigger an action based on the data sent by a client. A rule is composed of several parts. The first is the rule itself. It is defined by a name, an event type and a condition that will trigger this rule. Then, a rule can be composed of several actions that will be executed if the rule is triggered. POST /1.0/rules HTTP/1.1

```

208 Content-type: application/json
209 Authorization: Bearer JWTWithAdminRights
210
211 {
212   "name": "MahRoomRule",
213   "active": true,
214   "clients": [
215     2, 77, 196
216   ],
217   "eventtypeid": 4,
218   "function": "if((room == 'child' || room == 'kitchen') &&
219     temperature > 28)"
220 }
221 Response status 201:
222
223 {
224   "insertedId": 6
225 }
```

Code 19: Rule creation

As we have seen, a rule contains an event type. This will define which properties a client can send to the enclave and also define which properties we can use in our condition. In our case, the event type 44 allows a client to send the *room* and *temperature* properties. We can take these properties back into our condition and build a condition that says for instance that *if*

the temperature in the child's bedroom or kitchen is above 28, then an action is triggered. This rule can be activated by a certain list of clients only.

The second component of a rule is the list of actions. A rule can consist of one or more actions. When a rule is evaluated positively then it will trigger the actions associated with it. An action is composed of an action type, a function and a URL. The action type is used to specify the content of the body request that the enclave will make to the action target using its URL. The function allows the enclave to transform the event data sent by the client.

```
226 POST /1.0/rules/{ruleId}/actions HTTP/1.1
227 Content-type: application/json
228 Authorization: Bearer JWTWithAdminRights
229
230 {
231   "actiontypeid": 6,
232   "function": "return {
233     "from": "marcel.grosjean@unine.ch",
234     "to": alert@unine.ch,
235     "subject": "Temperature too high",
236     "content": "Alert the room {{room}} has a dangerous temperature
of {{temperature}} degrees celcius.",
237     "room": "{{room}}",
238     "temperature": "Math.round(temperature * 10) / 10"
239   },
240   "ruleid": 6,
241   "urlid": 3
242 }
243
244 Response status 201:
245
246 {
247   "insertedId": 7
248 }
```

Code 20: Rule action creation

An action is associated with a single rule and has a single action type and a single URL. The function property allows the rule engine to perform transformations on the data that will be sent to the action target. The action type gives us all the fields that are required and are as follows: *from*, *to*, *subject*, *content*, *room* and *temperature*. For the *from* and *to* fields, we have manually specified who is the recipient of the mail and its source. The subject is also entered manually and will be the same for all the actions that will be performed. The content field is interesting because we have put a templating system in place to replace variables with the values sent in the event. For example, *Alert the room {{room}}* will be evaluated and the room variable will be replaced by the value that will have been sent by the client. The action target will receive the following text: *Alert the room kitchen...* The *room* property tells us that it is possible to directly return the value of the event just by specifying the property name. It is also possible to make transformations on strings or numerical values. The temperature property shows an example of rounding a numerical value.

GET	/1.0/rules	findAllRules
POST	/1.0/rules	createRule
DELETE	/1.0/rules/{id}	deleteRule
GET	/1.0/rules/{id}	findRuleById
PUT	/1.0/rules/{id}	updateRule
POST	/1.0/rules/{id}/actions	createRuleAction
DELETE	/1.0/rules/{id}/actions/{idaction}	updateRuleAction
PUT	/1.0/rules/{id}/actions/{idaction}	updateRuleAction

Figure 16: Rules API illustration

3.5.10 Event

An event is the data measured by Things, then formatted according to an event type before being sent to the enclave.

```

249 POST /1.0/events HTTP/1.1
250 Content-type: application/json
251 Authorization: Bearer JWT
252
253 {
254     "clientid": 2,
255     "eventtypeid": 4,
256     "properties": {
257         "room": "kitchen",
258         "temperature": 29.145
259     },
260     timestamp: "2018-10-13T16:41:54.218Z"
261 }
262
263 Response status 201:
264
265 {
266     "insertedId": 8
267 }
```

Code 21: Event fired

As we can see, an event must be characterized by a client, an event type and a list of properties. The properties in an event and their data-type must match the properties specified in the event type. A timestamp that corresponds to the date time when the measurement was taken must also be specified. Indeed, the date time of the measurement may not be the same as the date time of sending the data.

Events are immutable messages that are sent once to the enclave. It is possible to add, view and delete them. However, it is not possible to modify them, which is why there is no PUT request. Once the data are sent to the enclave from the client, actions will be triggered if the rule engine decides to do so. Action messages are the results of actions done by the rule engine. It will therefore record the data sent to the action target and it is possible to get action messages per client.

```

268 GET /1.0/events/{clientId}/actionmessages?limit=10 HTTP/1.1
269 Authorization: Bearer JWTWithAdminRights
270 Response status 200:
271 Content-type: application/json
272 [{
273     "id": 9,
274     "eventid": 2,
275     "actiontypeid": 5,
276     "clientId": 2,
277     "completionTime": "2018-10-13T16:41:54.345Z",
278     "destclientId": 1,
279     "ruleid": 6,
280     "urlid": 3,
281     "message": "{\"content\": \"Alert, the room kitchen has a dangerous
temperature of 29.145 degrees celcius\", \"from\":
marcel.grosjean@unine.ch, \"to\": alert@unine.ch, \"subject\":
\"Temperature too high\", \"room\": \"kitchen\", \"temperature\": 29.1}
282 }]
```

Code 22: GET action messages

As we can see, action messages enable us to check that if an action has been performed then we can trace the source of the event, when it happened, the target and the content of the message.

GET	/1.0/events	findAll
POST	/1.0/events	createEvent
DELETE	/1.0/events/{id}	deleteEvent
GET	/1.0/events/{id}/actionmessages	getActionMessageModelByClientId

Figure 17: Events API illustration

3.5.11 Conclusion

We have seen in this section that it is possible to implement our own programming paradigm of events, condition and actions. This API was inspired by the iFlux project and adaptations were made to serve the needs of the project. It is therefore possible to process real-time data from events and initiate actions based on conditions without programming. The web interface allowing to manage the system is therefore an incredible added value for an enclave because no line of code would then be necessary in order to manage the enclave.

3.6 Enclave middleware

3.6.1 Introduction

In this section, we will get through all the technologies that have been selected for the technical implementation as well as the overall architecture of the enclave.

3.6.2 Spring boot

Spring is a very popular framework among Java developers for the many features it provides on web aspects, security, data access and much more. However, it is also known for its complex and tedious configuration. It is common to spend several hours/days configuring a Spring project. Spring developers then decided to work on a project to make it easier for developers to develop Spring applications and they created Spring boot.

Spring Boot is a micro framework that aims to facilitate the configuration of a Spring project and reduce the time allocated to start a project. The generation of a Spring boot project is done very quickly and either by a website (<https://start.spring.io/>) which allows to quickly generate the structure of a project with its Maven dependencies or by the Eclipse STS plugin (Spring Tool Suite). Auto-configuration, which applies a default configuration at the start of the application, simplifies configuration without restricting functionality. It can be enabled with the annotation `@EnableAutoConfiguration`. Spring Boot offers other advantages, especially in terms of application deployment. Usually, the deployment of a Spring application requires the generation of a .war file that must be deployed on an application server such as Apache Tomcat. Spring Boot simplifies this mechanism by offering the possibility to directly integrate a Tomcat server into a .jar executable. When it is executed, an embedded Tomcat will be automatically started to run the application [27].

There are several other popular web frameworks that are even easier to use than Spring Boot. We can for example cite Flask or Django which are frameworks in Python or even NodeJs for JavaScript. Their use is easier and faster than Spring Boot. However, Spring Boot, although more complicated to use, has a serious infrastructure behind it. It is used by large companies and has a community and a company that develops and perpetuates it. The documentation is awesome. Using Spring Boot means having the guarantee of a good compromise between performance, ease of use, stability and durability.

In this project, we will use Spring Boot with the *spring-boot-web-starter* dependency that allows us to quickly create a REST API. In the rest of this chapter, we will discuss the different components of our REST API.

The structure of the enclave is shown in Figure 18.

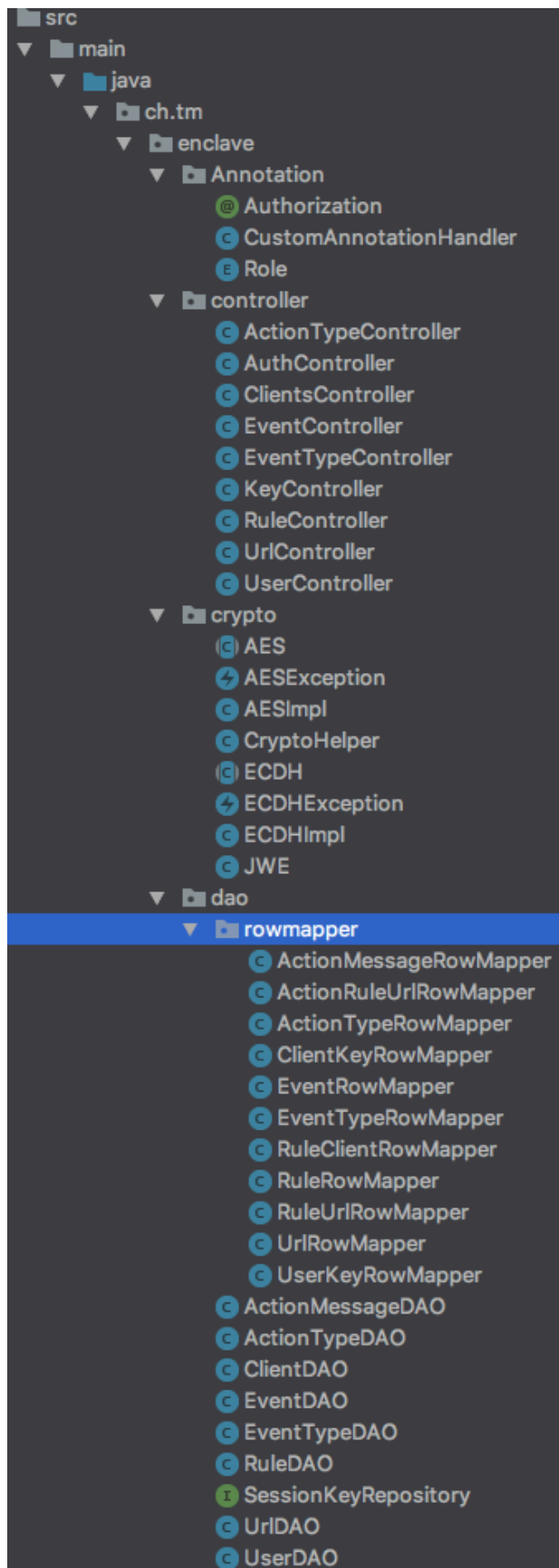


Figure 18: Enclave project structure part 1

The package *ch.tm.enclave.annotation* is here to check authorization at route level. It also contains the Role enum that defines all the roles.

The package *ch.tm.enclave.controller* contains all the controllers.

The package *ch.tm.enclave.crypto* contains all the cryptography functions. They are the same as in the gateway. The *ECDH* abstract class and *AES* abstract class are used for cryptography. The *JWE* class is a custom JWT generator class. The *CryptoHelper* file contains helpers for cryptography functions.

The package *ch.tm.enclave.dao* contains all the class that are used to access the database. No ORM are used here and the design pattern Data Access Object is used to access the object.

The package *ch.tm.enclave.dao.rowmapper* contains all the classes to transform relational data into Java POJO.

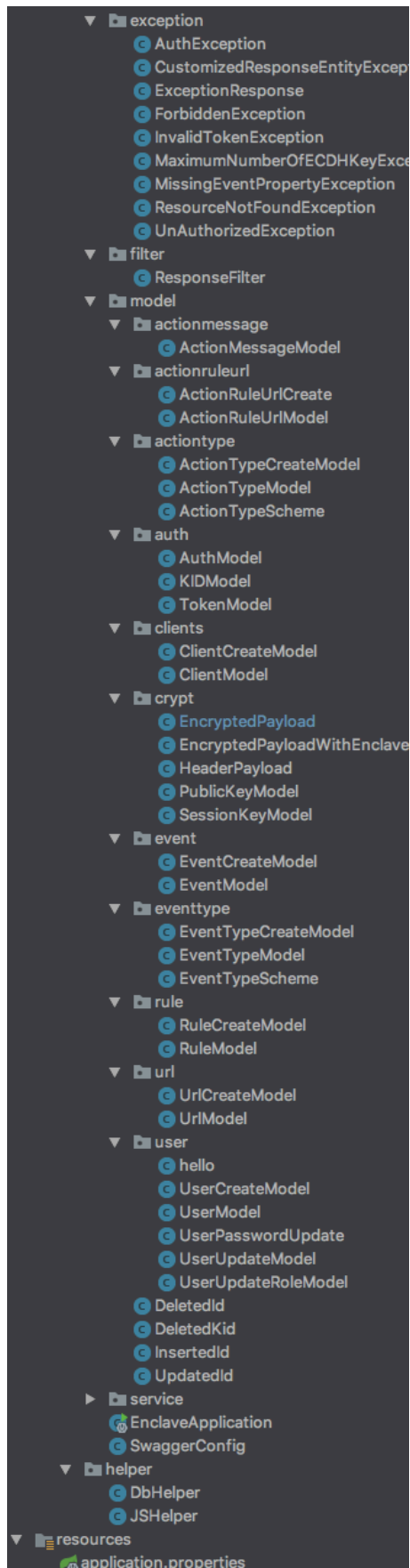


Figure 19: Enclave project structure part 2

The package *ch.tm.enclave.exception* contains the exception catcher and all the possible exceptions that can be thrown in the enclave.

The package *ch.tm.enclave.filter* contains the filter that will authenticate the user based on its JWT.

The package *ch.tm.enclave.model.** contains all the model used in the enclave. There are model for inserted|updated|deleted id. There are model that are used by the routes to check the validity of a request. For instance, the *event.EventCreateModel.java* is used to check the values of an event posted by a client. If the value in the request body does not match the model, then the request is invalid. The model *event.EventModel.java* inherits from *event.EventCreateModel.java* and adds an id. It's used as an event model to return events to the client.

The package *ch.tm.enclave.service* is the service layer used to access the Redis database.

The *EnclaveApplication.java* is the main Java file that run Spring Boot in autoconfig mode.

The *SwaggerConfig.java* file is used to configure swagger.

The *ch.tm.helper* package contains helpers functions.

3.6.3 Authentication/Authorization

It is important to distinguish between the two concepts because they are often mixed. Although both are related to security and access to shared resources, they are not similar. Authentication means that it is necessary to prove your identity to access a protected portion of a system. The authorization defines which resources an authenticated person can access. In other words, authentication defines who can access the system and authorization defines what resources an authenticated person can access [28]. To access the resources in the enclave, it is necessary to authenticate. All resources are protected except the following one: *POST:/1.0/auth*. For authentication, the request body must contain a valid login/password couple. Once accepted, the enclave generates a valid JWT with the class *ch.tm.enclave.crypto.JWE.java*. The generation of JWT is done without an external library. Code 23 shows an example of JWT.

```

283  B64-URL({
284    "alg": "dir",
285    "cid": <id of the client>,
286    "kid": <id of the secret key for this session>
287  }).
288  B64-URL({
289    "cid": "<id of the client>",
290    "mag": "<signature of the JWE>",
291    "admin": 0|1
292  }).
293  B64-URL("<JWT Signature>")

```

Code 23: JWT example after authentication

Once the login has been successfully completed, the client can access the resources. Authentication controls are done in the Filters. A Filter is called for each incoming and outgoing request. They can be used, for example, for redirection or permission checks. Code 24 shows an example of protection of all roads in the enclave, except login by a servlet Filter.

```

294  @Component
295  private class ResponseFilter implements Filter {
296    @Value("${jwt.secret}") private String secret;
297
298    @Override
299    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) {
300      String uri = (HttpServletRequest)request.getRequestURI();
301      if (uri.endsWith("/auth"){
302        JWE jwe = new JWE(secret);
303        String token = req.getHeader("Authorization");
304        if(!jwe.verifyTokenValidity(token))
305          throw new InvalidTokenException("This token is
invalid, expired or was not issued by us!");
306      }
307    }
308  }

```

Code 24: Servlet Filter example

In Spring applications, authorization is traditionally done in Filters. Indeed, each route is listed one by one in the Filter, and authorization checks are performed at this level. The Filter is applied to the entire API. If the user cannot access the resource, it is up to the Filter to return an error. In NodeJS applications, authorization checks are performed in middleware. A middleware in NodeJS is a function that is called before each route. It is the role of the middleware to check the authorization and send back an error message if needed. In a Flask API, authorizations are generally checked before each route is called with the use of annotations. Annotations are called before each route. The difference between Spring Boot and NodeJS/Flask is that the Filters are applied to the entire API while the middleware or annotations are applied individually for each route. Applying the authorization near the route allows more flexibility and it is not necessary to have a list of routes in the Filter.

There is not really a simple default mechanism to control route-level authorizations in Spring Boot. However, it is possible to create annotations and we therefore use the Flask mechanism as a model for creating our authorization mechanism. We therefore want to tag each route individually with an annotation. If an annotation is present, then it means that authentication is required. It is also necessary to specify a role (*user|admin*) in order to be able to authorize a certain type of user. Code 25 represents what the annotation would look like.

```
309 @Authorization(Role = Role.USER)
310 @GetMapping(path="1.0/auth/sayhello")
311 public String hello(){
312     return "{ 'hello': 'hello' }".replace("'", "\"");
313 }
```

Code 25: Example of protected route

We then need to create an enum of roles in Code 26.

```
314 public enum Role {
315     USER,
316     ADMIN
317 }
```

Code 26: Role enum

We then need to create an annotation handler that will be called before each route. The handler is in Code 27.

```

318  @Aspect
319  @Component
320  public class CustomAnnotationHandler {
321      @Value("${jwt.secret}") private String secret;
322      @Around("@annotation(Authorization)")
323      public Object Authorization(ProceedingJoinPoint joinPoint)
324          throws Throwable {
325          MethodSignature signature = (MethodSignature)
326          joinPoint.getSignature();
327          Method method = signature.getMethod();
328          Authorization auth =
329          method.getAnnotation(Authorization.class);
330          HttpServletRequest request = ((ServletRequestAttributes)
331          RequestContextHolder.getRequestAttributes()).getRequest();
332
333          String token = request.getHeader("authorization");
334          JWE jwe = new JWE(secret);
335          JSONObject claims = new JSONObject(claimsString);
336
337          // Check if the user is connected (has cid ?)
338          if (!claims.has("cid"))
339              throw new UnauthorizedException("You must be logged on
340              to access this ressource");
341
342          boolean admin = Boolean.valueOf(claims.getString("admin"));
343
344          // Then check if the user is authorized to be here
345          // If he's not admin we throw exception
346          if (auth.Role() == Role.ADMIN && admin == false)
347              throw new ForbiddenException("You are not allowed to
348              access this ressource");
349
350          // nothing bad happened ? Can now go further
351          return joinPoint.proceed();
352      }
353  }

```

Code 27: Spring Boot annotations handler

We have seen with the Filters that it is possible to control authentication at the level of the entire API. It is also possible to control the authorization, but it is less flexible. We took Flask and its annotation system to create our own authorization system in order to provide a very flexible mechanism. It should be noted that we can do authorization and authentication directly in the annotations without Filter.

3.6.4 Errors Handling & Logging

Handling errors correctly in a REST API and returning a clear and explicit error message to the user is a very desirable feature for both for the user who immediately understands what is happening and for the developer who immediately knows the cause of the error [29]. The default behavior of Spring Boot is to return the stack trace of the error to the user. It is neither user-friendly nor very secure to do so. A good way is to return a simple and clear error to the client and log the exception in an error file that the developer looks at the details later.

Spring Boot has a very easy and efficient error handling mechanism. It is based on exceptions that are thrown in the application without being caught. These exceptions are then caught by the *ResponseEntityExceptionHandler*, which will process them. Code 28 is an example of some exception handling for invalid SQL queries as well as authentication errors.

```
348 @ControllerAdvice
349 @RestController
350 public class CustomizedResponseEntityExceptionHandler extends
    ResponseEntityExceptionHandler{
351     private static final Logger logger =
        LoggerFactory.getLogger(CustomizedResponseEntityExceptionHandler.cl
            ass);
352
353     @ExceptionHandler(SQLException.class)
354     public final ResponseEntity<Object>
        handleUniqueException(Exception ex, WebRequest request){
355         logger.error(ex.getMessage());
356         if (ex.getMessage().contains("Duplicate"))
357             return new ResponseEntity(new ExceptionResponse(new
                Date(), "Duplicate entry", HttpStatus.CONFLICT.value(),
                request.getDescription(false)), HttpStatus.CONFLICT);
358
359         return new ResponseEntity(new ExceptionResponse(new Date(),
                ex.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR.value(),
                request.getDescription(false)), HttpStatus.INTERNAL_SERVER_ERROR);
360     }
361
362     @ExceptionHandler(UnAuthorizedException.class)
363     public final ResponseEntity<Object>
        notAuthorizedAccess(Exception ex, WebRequest request){
364         logger.error(ex.getMessage());
365         return new ResponseEntity(new ExceptionResponse(new Date(),
                ex.getMessage(), HttpStatus.UNAUTHORIZED.value(),
                request.getDescription(false)), HttpStatus.UNAUTHORIZED);
366     }
367 }
```

Code 28: Spring Boot exceptions handler

The *SQLException* and *UnAuthorizedException* classes must exist and extend either *Exception* or *RuntimeException* class. As we can see, SQL or authorization errors are caught, the message is logged in a file and a clear error message is sent to the client. It should be noted that *SQLExceptions* are difficult to debug because no matter what type of exception, an SQL exception will always return *SQLException* with a message inside. To know the type of error, it is then necessary to look at the content of the message. In our case, we catch the message and check if it contains the "Duplicate" word in order to see if it is a unique key constraint violation type error. Code 29 shows an example of a message returned to the client.

```
368 GET /1.0/events HTTP/1.1
369 Response status 401:
370 Content-type: application/json
371
372 {
373     "timestamp": "2018-10-15T23:58:47.872+0000",
374     "status": 401,
375     "error": "Unauthorized",
376     "message": "Unauthorized",
377     "path": "/events"
378 }
379
```

Code 29: Example of error returned by the API

3.6.5 Database type & model

The database is necessary to store all the data that are useful for running the rule engine. Clients making requests are stored as well as action targets. Events sent to the enclave are also stored. The rule engine and all properties and actions associated with it are also stored. The results of the actions are stored there. We see here that all data used to manage the rule engine as well as all incoming events and outgoing actions are stored. We keep a complete history of incoming and outgoing data.

The problem here is what kind of database it is useful to use. It is indeed possible to use a relational database or NoSQL database and particularly a document-oriented database. Indeed, they have their strengths and weaknesses. Relational databases are very effective at managing structured data and have a powerful query language (SQL). However, they sometimes have trouble for making complex queries as soon as the amount of data becomes very large. For example, we can imagine a join on several tables, each of which has a large amount of data. Non-relational databases have the ability to store large amount of unstructured data and can be easily scalable [30]. Being able to store unstructured data is particularly useful for storing data from heterogenous environments. A document-oriented database would be particularly useful for storing data from sensors.

We have seen that we have two types of data to manage. We have the data of the rule engine that is structured and in small quantity. Indeed, the rule engine is composed of rules, event types, action types and actions. These are relational data that are perfectly adapted for SQL language. The amount of data to manage by the rule engine is very modest. Then the second type of data are event data and action messages. Events are data sent by sensors and can be sent in very large quantities. This data does not correspond to any pre-defined schema in the database. They are formatted according to the event type and then sent. As a result, there is a large amount of unstructured data coming to the database. Then there are the action messages. These are the data that results from the actions made by the rule engine. They comply to the action type and are unstructured. They can also be found in very large quantities in the database.

The first solution to solve this problem is to decide to use only a document-oriented NoSQL database to store events, action messages and the rule engine. This is quite possible and very efficient. The advantage is that since most data are unstructured, performance will be very good

and it will be very easy to store unstructured data. However, we lose the SQL query language that is useful to us in order to retrieve data from our rule engine and making complex requests.

The second solution is to use a mixture of relational and non-relational databases. Thus, we would store events and action messages in a document-oriented database and the rule engine in a relational database. We would therefore have the advantage of both types of databases. However, this is not the chosen solution because it involves managing two different databases with their maintenance/updates chores and failure probabilities. Moreover, as soon as it is necessary to make a join between the data in the two databases, it immediately becomes less easy and less efficient.

The last solution is to use a relational database only. Thus, we store the rule engine and the events and action messages in the same database. This is the chosen solution because we keep SQL for the rule engine and event data are mainly intended to be inserted. The cost of queries will be high to make a join for selection but reduced by the fact that selection queries are less common in our context. Figure 20 illustrates the implemented data logic model.

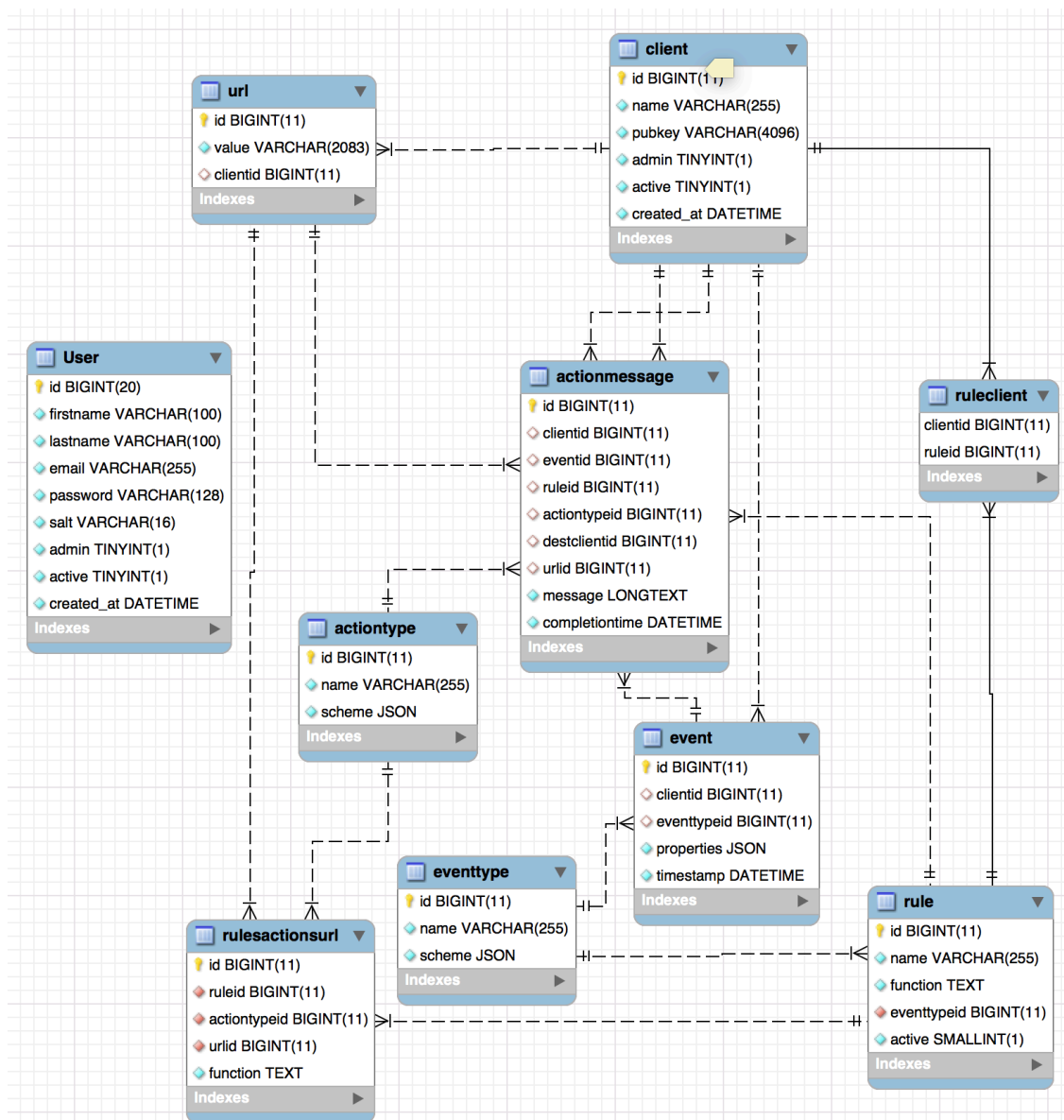


Figure 20: Logic data model

As we can see, the *events* and *actionmessages* tables store events and action results. These tables can be made up of a very large amount of data and queries can take a considerable amount of time. However, these tables are mainly intended to be a storage space and selection queries may be a limited part. Joins are generally avoided in order to keep reasonable performance and selection on id (primary or foreign key) are done with indexes, which makes selection queries fast. We can also notice that the data stored in the *event* and *actionmesages* table are unstructured. That is why the JSON data type was used to store data from Things. This makes possible to store unstructured data in JSON format in a relational database.

The *eventtype* and *actiontype* tables are mainly used by the rule engine. The data in the schemes are unstructured and that is why the JSON data type was used. The *rule*, *url*, *client* and *rulesactionurl* tables are much more classic and are mainly used to manage the rule engine.

The relational database chosen is Oracle MySQL because it has all the necessary functionality to develop our enclave. In addition to being free, it is powerful, can store a reasonable amount of data and has a tremendous community [31]. We have seen here that it is possible to store in a relational database, small data with large amounts of data without loss of performance while keeping the power of the SQL language. Unstructured data is managed with the JSON data type which also gives us great flexibility.

3.6.6 Database Access

Another important point to be addressed is the access to the data from the enclave. Spring Boot offers Spring Data. The latter is an abstraction layer for Spring JDBC that allows easier access to data without having to write a huge amount of redundant code and without having to manage transactions. Spring Data is in charge of object-relational mapping, transaction management and basic CRUD queries by offering a very high-level API based on the Domain Driven Design pattern (DDD) [32]. It is then no longer necessary to write code to do the basic CRUD operations and if we need to make more complex requests, which happens very often, it is always possible to do an object-relational mapping with Spring Data JPA and to make requests in JPQL [32]. Having a very high-level API allows us not to be locked by a certain database vendor, so database modification or migration is relatively easy.

DAO (Data Access Object) is a design pattern that provides an abstraction layer to communicate with a database without having to know the implementation details. A DAO manages SQL queries, transactions and directly uses the Spring JDBC driver without going through a higher-level abstraction layer. The advantage is that it is easy to make requests that correspond exactly to our needs and that is very efficient in terms of speed and memory utilization. The disadvantage is that the implementation is done by the developer and there is therefore a greater risk of errors. If the database changes, it is necessary to modify the source code accordingly. DAO locks the implementation to a certain database vendor which makes migration difficult.

Even though Spring Data offers an API allowing very easy access to the data, the solution retained remains the DAO pattern. Indeed, given the relatively small amount of different CRUD requests to be made and their complexity, it is much easier to implement them with a DAO.

Data validation at the REST API level is done with domain model classes. It is possible to reuse these classes with the DAO for CRUD operations. In addition, the nature of this project requires optimal performance and the DAO provide direct access to Spring JDBC, which makes queries faster by bypassing the various abstraction layers.

Code 30 shows an example of a DAO for the selection of all clients.

```
380  @Service
381  public class ClientDAO {
382
383      @Autowired
384      private NamedParameterJdbcTemplate jt;
385
386      public List<ClientModel> findAll(){
387          String sql = String.join("\n", "select id, name, pubkey,
admin, active, created_at ",
388                                          "from client ",
389                                          "order by name asc");
390
391          return jt.query(sql, new ClientKeyRowMapper());
392      }
393  }
```

Code 30: DAO example

This example without transaction shows how simple it is to make a relational-object mapping with DAO. The *ClientModel* class is a simple JavaBean that was created to validate the data posted in the REST API and was reused with the DAO. The *ClientKeyRowMapper* class is a class used to map the fields of the SQL query into a POJO. Code 31 is an example of *KeyRowMapper*.

```
394  public class ClientKeyRowMapper implements RowMapper<ClientModel> {
395      @Override
396      public ClientModel mapRow(ResultSet rs, int rowNum) throws
SQLException {
397          return new ClientModel(
398              rs.getLong("id"),
399              rs.getString("name"),
400              rs.getString("pubkey"),
401              rs.getBoolean("admin"),
402              rs.getBoolean("active"),
403              DbHelper.dateTimeFromString(rs.getString("created_at"))
404          );
405      }
406  }
```

Code 31: KeyRow mapper

3.6.7 Rule engine

Introduction

The rule engine is an essential part of the enclave because it allows to perform actions on events data based on rules. A client sending data must ensure that the data matches the specified event type. Once the data are received, the enclave must infer on it using its rule engine. The events are then evaluated according to the rule condition of the rule associated with the client and the event type. The rule condition is a string hard coded by the enclave administrator. If this condition is true, then the enclave must perform the actions that are associated with this rule. An action consists in formatting a message according to an action type. Data transformation can be performed. Once the action message is created, it is then sent to the action target. All events are recorded as well as all actions performed.

Figure 21 shows how the rule engine works schematically. Cryptographic functions are not considered here.

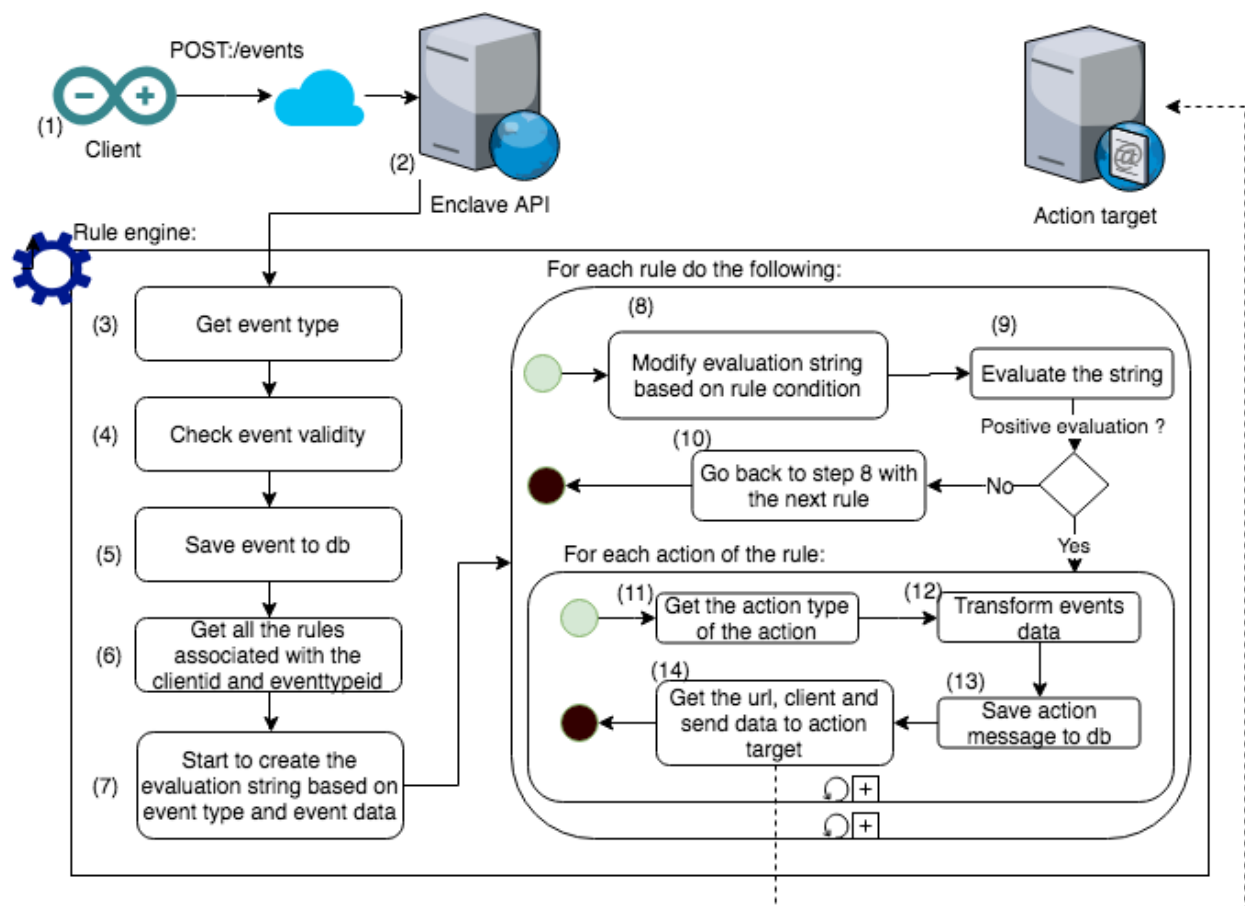


Figure 21: Rule engine schematic illustration

1. A client gathers data and makes an HTTP request to the enclave at the URL *POST:/1.0/events* with its *clientid*, its *eventtypeid* and the corresponding data.
2. The enclave controls the validity of the request but not the content. If the request is valid, then it sends the data to the rule engine.
3. The rule engine gets the event type from the database in order to control that the request really complies with the event type.
4. The rule engine checks if the request complies completely with the event type. If not, it returns an error message to the client.
5. The event data are saved to the database.
6. The rule engine gets all the rules associated with this client id and event type id.
7. The rule engine starts to create a string that will be evaluated in order to trigger actions. This string is made of variables that come from the event and event type.
8. For all rules, we associate the rule condition to the evaluation string.
9. There are two possible outcomes for the evaluation. The first is that the condition is not met then we skip to the next rule. If the condition is met, then we start the action.
10. If the condition is not met, we start over with the next rule.
11. If the condition is met, we first get all the actions for this rule and the action type for each action.
12. For all actions, the rule engine transforms the data in the events and create a message that will be sent to the action target. The message must comply with the action type.
13. The rule engine saves the action message in the database.
14. The rule engine gets the URL of the action target and make a HTTP request containing the action message. Once it's done, the rule engine makes the same for the next action.

As we can see, the rule engine performs very complex tasks based on unstructured data. Indeed, the event data are structured according to an event type. The condition that allows an action to be executed is in the form of a string of characters and the actions and transformations to be performed are also in the form of a string. In order to run the rule engine, we need to find a way to read, parse and evaluate string and take actions based on the result.

After careful consideration, it was decided to use JavaScript to dynamically build code that could be read, parsed and executed by the rule engine. The code built is based on events, event types, rules condition and action types. This code is then evaluated by the Java's JavaScript execution engine to run our string-based rule engine. This is how the rule engine works in detail:

Practical example

In this part we will cover a practical use case and see how the rule engine process the event.

Step 1: Code 32 shows a client that gather data and send content to the enclave.

```
407 POST /1.0/events HTTP/1.1
408 Content-type: application/json
409 Authorization: Bearer JWT
410
411 {
412   "clientid": 2,
413   "eventtypeid": 4,
414   "properties": {
415     "room": "kitchen",
416     "temperature": 29.145
417   },
418   timestamp: "2018-10-13T16:41:54.218Z"
419 }
```

Code 32: Send event for rule engine

Step 2: The enclave validates the event and send the data to the rule engine.

Step 3: The rule engine gets the event type n°2 from the database. Code 33 represents an event type.

```
420 {
421   "name": "TempEventType",
422   "scheme": {
423     "properties": {
424       "room": {
425         "type": "string"
426       },
427       "temperature": {
428         "type": "number"
429       }
430     },
431     "required": [
432       "room", "temperature"
433     ],
434     "type": "object"
435   }
436 }
```

Code 33: Event type example for rule engine

Step 4: The rule engine turns the event type JSON scheme string into a readable Map structure. Then checks one by one if all the required properties are in the event and if the type of the property match. In our case they correspond, and we can proceed.

Step 5: The event is valid; the rule engine saves it to the database.

Step 6: The rule engines get all the rules associated with this client id and event type id. Code 34 show all the actions for this specific rule.

```
437  [{
438    "name": "MahRoomRule",
439    "active": true,
440    "clients": [
441      2, 77, 196
442    ],
443    "eventtypeid": 4,
444    "function": "if((room == 'child' || room == 'kitchen') &&
445      temperature > 28)"
446  ]}
```

Code 34: Actions for specific client and event type

Step 7: The rule engine starts to create a string that will be evaluated in order to trigger actions. This string is the JavaScript code in Code 35.

```
446  // theses variables come from the event type
447  var room;
448  var temperature;
449  // theses variables are set with the values from the event
450  room = 'kitchen';
451  temperature = 29.145;
```

Code 35: Basic string for evaluation

Step 8: For all rules, we associate the rule condition to the evaluation string. The JavaScript code is in Code 36.

```
452  var room;
453  var temperature;
454  room = 'kitchen';
455  temperature = 29.145;
456  if((room == 'child' || room == 'kitchen') && temperature > 28)
457    print('true'); // this will trigger the rule
458  else
459    print('false'); // this will NOT trigger the rule
```

Code 36: Script that will trigger an action

The if condition changes according to the rule. The code above is evaluated by Java's JavaScript engine and if the result is true, then the actions of the rules can be triggered.

Step 9: For each valid rule, we get all the actions associated with this rule. They can be found in Code 37.

```

460  [{
461    "actiontypeid": 6,
462    "function": "return {
463      "from": "marcel.grosjean@unine.ch",
464      "to": alert@unine.ch,
465      "subject": "Temperature too high",
466      "content": "Alert the room {{room}} has a dangerous temperature
of {{temperature}} degrees celcius.",
467      "room": "room",
468      "temperature": "Math.round(temperature * 10) / 10"
469    }",
470    "ruleid": 6,
471    "urlid": 3
472  }]

```

Code 37: Actions of a rule

Step 10: For each action, we get the action type associated in Code 38.

```

473  {
474    "name": "TempActionType",
475    "scheme": {
476      "properties": {
477        "from": { "type": "string" },
478        "to": { "type": "string" },
479        "subject": { "type": "string" },
480        "content": { "type": "string" },
481        "temperature": { "type": "number" },
482        "room": { "type": "string" }
483      },
484      "required": [
485        "from", "to", "subject", "content", "temperature", "room"
486      ],
487      "type": "object"
488    }
489  }

```

Code 38: Action type of a rule's action

Step 11: The rule engine turns the function property from **step 9** into a readable Java JSONObject with the Code 39 example.

```

490  String f = arm.getFunction().substring(8, arm.getFunction().length()
- 1).trim();
491  JSONObject dict = new JSONObject(f);

```

Code 39: Turn String JSON into Java JSON Object

The *arm* variable contains the action for this rule as a Map.

Then the rule engines extract the scheme from **step 10** and turn it into a Java Map in Code 40.

```
492 Map<String, Map<String, String>> scheme =  
    etm.getScheme().getProperties();
```

Code 40: Turn String scheme into a Java Map

The variable *etm* contains the action type for this action.

Step 12: So far, we know that this rule is valid. We have all the actions for this rule and the action type associated. We now need to apply data transformation in order to create the action message. To do so, we parse the action type and for each property of the action type, we will create a new JSON object containing the property with the data from the event.

In summary: For each property of the action type, we create a new JSON object with all the action type properties. For each property of the new JSON document, we set the value as the same as in the action. And for each action value, we check whether it's a string or a number/integer.

If it's a string, we check with a regex if there is an expression like `{{variable}}` and we replace this variable with the variable from the event. It's a templating process. If it's a number/integer, we apply data transformation with a valid JavaScript expression.

Code 41 illustrates the whole process.

```

493 Map<String, String> actionResult = new HashMap<>(); // final result
494 //Now parse the function dict in order to evaluate the result
495 for (String key : dict.keySet()){
496     String e;
497     // Property is string
498     if
        (actionTypeProperties.get(key).get("type").toLowerCase().equals("string")){
499         String finalString = dict.get(key).toString();
500         try {
501             Pattern pattern = Pattern.compile("\\\\{2} *[a-zA-Z0-9]+
        *}{2}");
502             Matcher matcher = pattern.matcher(dict.get(key)
        .toString());
503             while (matcher.find()) {
504                 String rawk = matcher.group().toString();
505                 String k = rawk.substring(2, rawk.length() -
        2).trim();
506                 finalString = finalString.replace(rawk.trim(),
        props.get(k).get("value").toString().trim());
507             }
508         }
509         catch (Exception ex){}
510
511         e = evalStr.toString() + "print('" + finalString + "') ; ";
512     }
513     else // Property is number/integer
514         e = evalStr.toString() + "print(" + dict.get(key) + ") ; ";
515
516     String evalResult = JSHelper.evalJS(e).toString();
517     actionResult.put(key, evalResult.trim());
518 }

```

Code 41: Data transformations done by the rule engine

The *actionResult* variable is the action message. The *evalString* variable is the content of **step 7**. The *e* variable contains the JavaScript code that will be evaluated by the *JSHelper* execution engine.

If the property is a string, it will try to find variables in brackets with a regex. Code 42 shows an example of templating.

```

519 Alert the room {{room}} has a dangerous temperature of
    {{temperature}} degrees celcius.

```

Code 42: Example of templating

The rule engine will find the variables in bracket and replace the values by the variables in the event. The final result looks like in Code 43.

```

520 Alert the room kitchen has a dangerous temperature of 29.145 degrees
    celcius.

```

Code 43: Templating done

If the property is a number/integer, it will execute the instructions in Code 44 in order to apply data transformation.

```
521  var room;
522  var temperature;
523  room = 'kitchen';
524  temperature = 29.145;
525  print(Math.round(temperature * 10) / 10);
```

Code 44: Data transformation on numerical values

The printed temperature will be set as the value of the property temperature.

And finally, once all the step done, the action message looks like in Code 45.

```
526  {
527    "content": "Alert, the room kitchen has a dangerous temperature
of 29.145 degrees celcius",
528    "from": marcel.grosjean@unine.ch,
529    "to": alert@unine.ch,
530    "subject": "Temperature too high",
531    "room": "kitchen",
532    "temperature": 29.2
533  }
```

Code 45: Final action message

Step 12: The rule engine saves the action message in the database.

Step 13: The rule engine gets the URL and client for this action and send an HTTP request to the action target with the action message in the body. Once the request made, it can proceed to the next action.

Conclusion

As we have seen, our JavaScript engine allows us to build JavaScript expressions dynamically according to events, rules, actions and action types. We have implemented the events, condition, action paradigm with our rule engine.

3.7 Action Target

The action target is called by the enclave to perform an action. Once the rule engine has identified an action to trigger, it gets the URL of the action target, then puts in the request body the action message that corresponds to an action type. This action type must be identical to the validation parameters defined in the action target REST API. Each route of the action target must be accessible from the enclave.

The action target is an actuator that can implement several types of actions. We can imagine push notifications, emails, SMS or a physical actuator such as a servomotor or an electric car. For the sake of simplicity, in this project, we stopped at the implementation of an email sending interface.

The language used to implement the action target is NodeJS with the Express framework. The API is documented with Swagger, errors are logged in a logfile and we use Gmail as the mail server. It also uses handlebars.js as a mail templating system.

Figure 22 shows an example of 3 routes to send email:

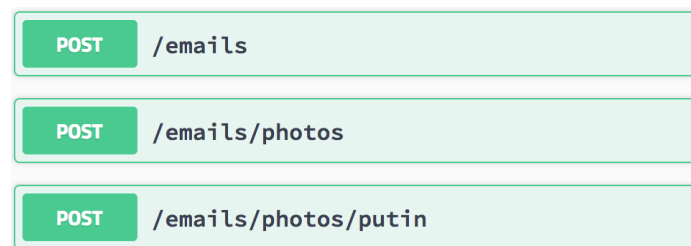


Figure 22: Action target API illustration

The */emails* route is implemented in NodeJS with Code 46.

```

534 app.post('/emails',
535   expressJoi({body: {
536     sender: Joi.string().email().max(255).required(),
537     dest: Joi.string().email().max(255).required(),
538     subject: Joi.string().min(1).max(4096).required(),
539     content: Joi.string().required()
540   }}),
541   async (req, res) => {
542     let sender = req.body.sender.trim().toLowerCase();
543     let dest = req.body.dest.trim().toLowerCase();
544     let subject = req.body.subject.trim();
545     let body = req.body.body.trim();
546
547     let mailOptions = {
548       from: sender,
549       to: dest,
550       subject: subject,
551       template: 'template',
552       context: {
553         bodycontent : body,
554         putin: false,
555         complex: false
556       }
557     };
558     try {
559       let result = await transporter.sendMail(mailOptions);
560       console.log(result);
561       res.status(204).json();
562     }
563     catch (err){
564       logger.error(err);
565       res.status(504).json();
566     }
567   });

```

Code 46: Action target example with NodeJS

As we can see, the route in Code 46 takes the following properties in the body: *sender*, *dest*, *subject* and *body*. The action type of the enclave must comply with the data validation of the middleware. The action message of the enclave must, of course, also comply with it, otherwise, an error 400 is returned by the action target. Code 47 query shows an example of mail that can be sent.

```

568 POST /emails HTTP/1.1
569 Content-type: application/json
570
571 {
572   "from": "marcel.grosjean@unine.ch",
573   "to": alert@unine.ch,
574   "subject": "Temperature too high",
575   "content": "Alert the room kitchen has a dangerous temperature
of 29.145 degrees celcius."
576 }
577 Response status 204:

```

Code 47: Example of action message for sending an email

3.8 Front-end (RIOT)

3.8.1 Introduction

It was decided to call the front-end *RIOT* (Responsive Internet of Things). Giving a name to a project gives a better and clearer vision for the rest of the project development.

We want to create a modern front-end that uses cutting-edge technologies. Single Page Applications (SPA) are JavaScript framework that allows developers to use the latest web technologies in order to develop rich web applications. The problem with modern frameworks is that there is an impressive amount of them on the market and it is very difficult to say at the moment which framework will be the most appropriate for our needs and especially which one will offer the best longevity in terms of update and community support. The popularity of JavaScript as a client-side and server-side programming language has increased from year to year. JavaScript 6 (ECMAScript 6) has become a must for Single Page Applications. It allows to quickly prototype a web application and lets developers focus on functionality rather than code structure. The advantages of using JavaScript frameworks are multiple. The first is efficiency. Indeed, since frameworks come with a predefined structure, and tools to generate and compile code, it is then easier for developers to focus on features, which advances the development speed. Security is another argument for SPAs because they are tested by a large community to quickly identify bugs and security vulnerabilities. Moreover, the frameworks are free, however, the hidden price is the learning time necessary to be able to be operational with these frameworks. Indeed, some are known to have a fairly steep learning curve.

In order to choose a framework, we will base our choice on the popularity of different frameworks in 2018. And we will arbitrarily choose the most popular framework on GitHub. Figure 23 shows the popularity of the different frameworks in 2018.

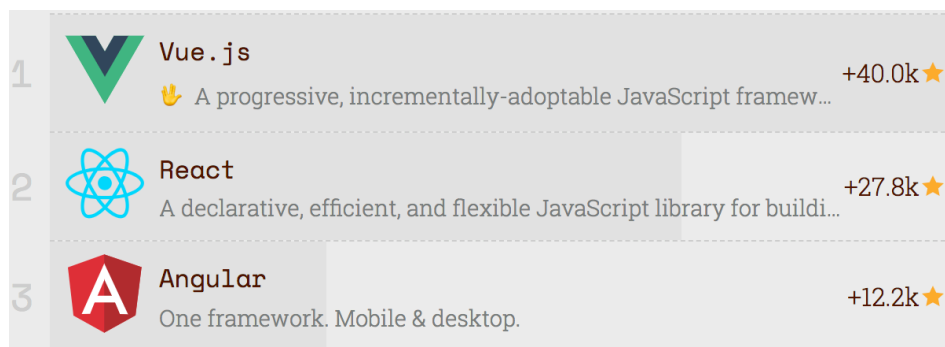


Figure 23: Front-end Framework ranking 2018²

As we can see in the illustration, the most popular framework on GitHub at the moment (2018) is Vue.js. Therefore, we will rely on popular wisdom and use Vue.js for our project.

² <https://risingstars.js.org/2017/en/#section-framework>

Vue.js is an open source JavaScript front-end framework for creating web GUIs. It simplifies development by better organizing the code and separating the different components. The main concept of Vue.js is to develop reusable components that are the combination of JavaScript and HTML template code. These components have a weak coupling between them and encourage reusability throughout the application. Vue.js has several advantages, such as size. Indeed, the strict minimum of Vue.js has a size of 21KB. It is also very affordable in terms of complexity, and comes with a very clear documentation, as a result it is easy to develop a prototype quickly [33]. Vue.js is therefore an appropriate choice because it provides us with all the tools to develop a SPA as well as normally a long-term support because it has a large community.

The other undeniable advantage that comes with Vue.js is that if we use all the generation tools, then the deployment becomes very easy. During development, we will use all the features of ES6 that are not browser compatible. We will then use the webpack tool which will transpile our ES6 code into browser-compatible JavaScript code. It will also minimize the files and obfuscate them in order to make them as light and unreadable as possible. The final code are HTML, CSS and JavaScript files that we will have to put on a CDN in order to serve them. We therefore do not need a backend to be able to run our application. No backend other than the enclave is required to run the web interface.

Another challenge is to develop an application that is both effective and aesthetically pleasing. Indeed, developers are not always the best prepared to develop applications with a devastating look. That's why they need to use a pre-made user interface kit that is nice looking and that has been designed to be effective. Material Design was developed by google and offers a unified graphical user interface that allows us to create digital experiences with proven techniques and design [34]. Material Design is a set of rules, procedures, components and best practices to create an application and above all a user experience. Google created Material Design to unify the look of all their applications. We can therefore reuse their work in order to create our own application without having to define our own visual identity. Another advantage of using Google's design is that the end user will feel that the quality will be the same as an application made by Google itself. The disadvantage is that all applications will look the same.

In this project, we will use the Vuetify.js framework which is an implementation of Material Design made for Vue.js. It is free, and its documentation and community are very active and well done.

3.8.2 Use cases

Figure 24 shows the global use case schema for the front end.



Figure 24: RIOT use cases diagram

As we can see, we have three external users:

- **Client:** A client is any Thing that has an open session and is logged in.
- **User:** A RIOT user is a client with administrator rights. A semantic distinction is made to separate the client that is used to send events from the user that does the same thing but in addition can manage the whole system.
- **Action target:** An action target is a micro service that we call after a rule has been triggered during an event.

To be able to use the platform, a client must first create a session with the key exchange protocol. Once the session is created, the client can log in with his email/password combination. The only action a client can do is to send events to the middleware. These events will be recorded and then processed by the rule engine. If a rule is triggered, then all actions related to that rule will be fired. The actions taken are recorded.

A difference was made between clients and users in RIOT. Indeed, a user is actually a client with the administrator role. A distinction has been made in RIOT because a user who will

manage the system will do so with an email/password credential. Once the user has logged in, he can then use the platform as a client would with administrator rights.

A user manages the different event types and action types. It defines the rules to be applied when an event occurs. It also defines which URL to call and actions to be taken in case of a positive evaluation by the rule engine. It also manages clients, users and cryptographic keys. It can also see all the events that have been sent and the actions that have been triggered.

3.8.3 Navigation Diagram

Figure 25 shows the different pages of the web application with the transitions between all the pages. All use cases are implemented. Navigation begins with the login page once a session has been established. Then the user has access to the whole application if he is an administrator user, otherwise only the green part will be accessible.

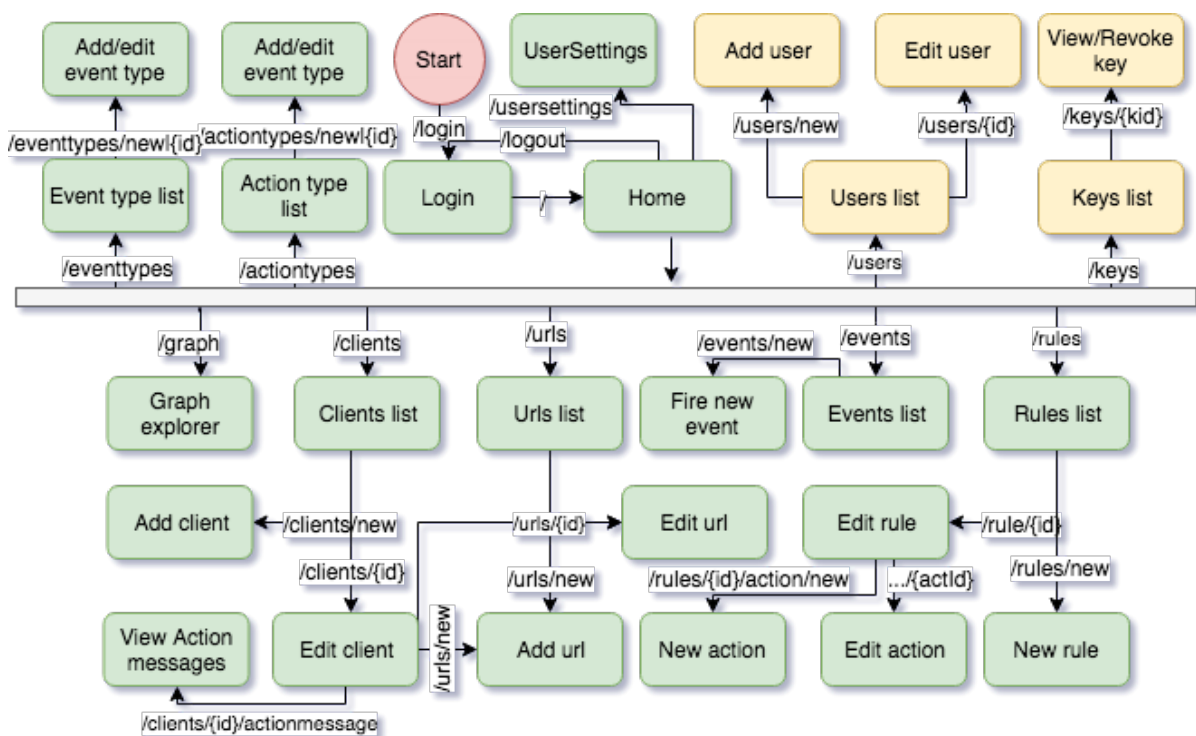


Figure 25: RIOT navigation diagram

The RIOT project is also internationalized (i18n) and for now, the only language that is supported is English, but the project is translation-ready.

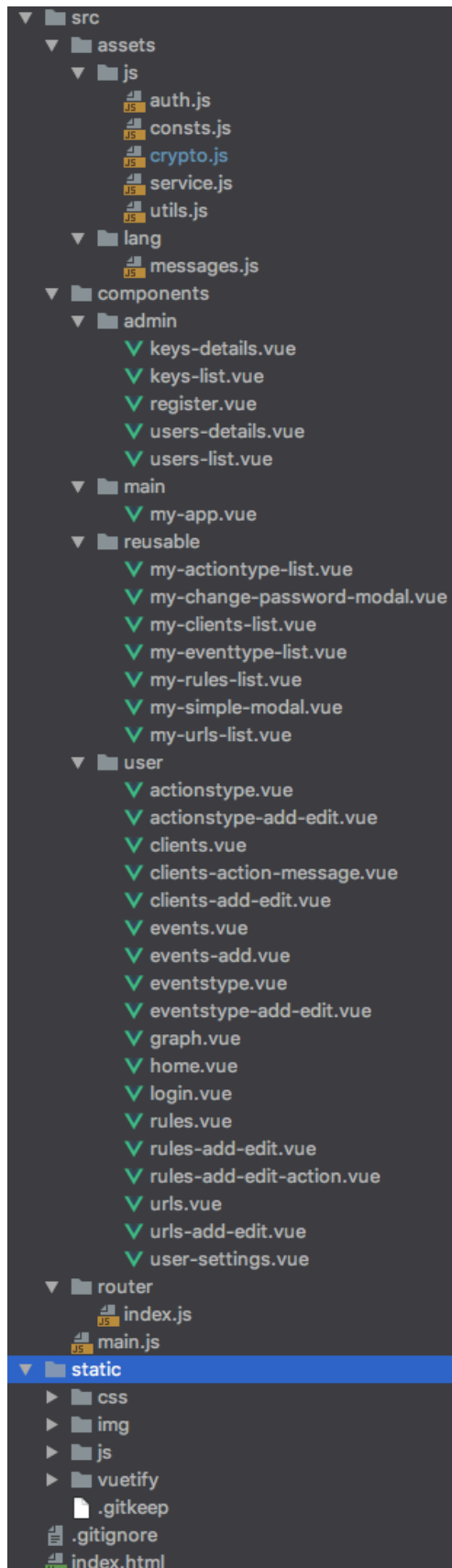


Figure 26: Front-end project structure

Figure 26 shows the description of the structure of the front end.

The *index.html* file is the base file of the vue HTML part. The *main.js* file is the main file for vue configuration. The *router/index.js* file contains all the routes and authorization to access the page.

The *components/* folder is where all the vue components are located. A vue component is the mix of HTML elements and JavaScript code. The main component where all the other components are displayed is the *components/main/my-app.vue* file. The components in *components/reusable* are registered as reusable across the whole vue app.

The *assets/* folder contains all the JavaScript code that can be imported in the project and that will be compiled with the rest of the project. The *assets/lang/messages.js* file contains all the i18n strings. The file *assets/js/auth.js* and *assets/js/crypto.js* contains all the functions that manage authentication and cryptography. The *assets/js/service.js* file is a service layer. The *assets/js/utils.js* and *consts.js* files are constants and functions helpers.

The *static/* folder contains all the external dependencies such as some JavaScript, CSS, and images.

Once the project is compiled, the executables are moved in the *dist/* folder.

3.9 Practical example of cryptographic functions

3.9.1 Introduction

In this section, we will use the theoretical concepts presented in the theoretical part as well as the practical concepts presented in this part to present a practical use cases using all the mechanisms described. The aim is to present the interesting parts of the code and the essential concepts that will be illustrated in a practical example. We will cover the different mechanisms used to create a session, login and exchange messages.

3.9.2 Session Key Creation

Communication between clients and the enclave is secured. The desire to make encryption transparent from the developer's point of view has led to the creation of several components necessary for their implementation.

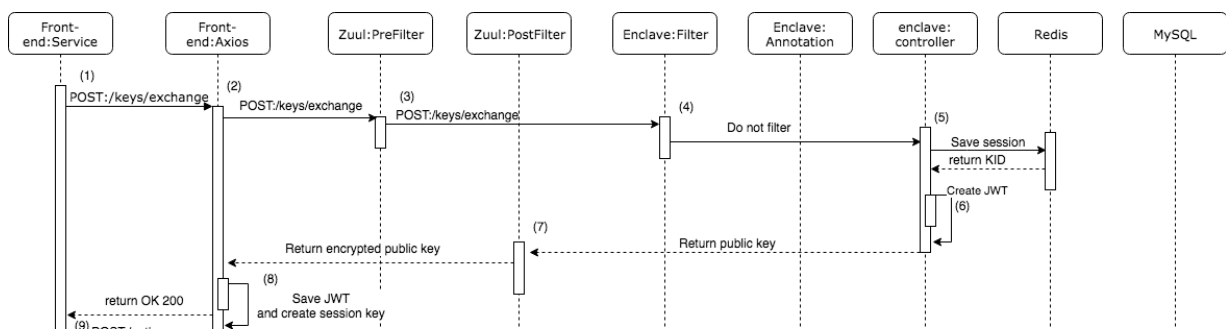


Figure 27: Session establishment sequence

This UML sequence diagram in Figure 27 shows the steps required to create a session between a client and the enclave. We assume that they have never exchanged messages before and that session creation is the first step.

Step 1

In our web interface, all HTTP communications are done through a service layer. The first step is to exchange with the enclave their respective public keys.

```
578 POST /1.0/keys/exchange HTTP/1.1
579 Content-type: application/json
580
581 {
582     "publickey": "3059301306072A8648CE3D020106082A8648CE3D03010703420
004809CB1C845DF75A504C6B1AC03F33D139DA99EEEA3E140433983C4D61CCF1D42CF25A7
F296816ABB7B49AA644FAAF3E5FC19A632C66EA764CA3E18B71FAEBE68"
583 }
```

Code 48: Example of session creation

Step 2

The request is intercepted by Axios. The only URL that is not encrypted is *POST:/1.0/keys/exchange*. Axios will send the request as it is.

Step 3

The request is intercepted by Zuul pre-filter. It sends the request as it is.

Step 4

The request is intercepted by the enclave filter. It sends the request to the controller as it is.

Steps 5 & 6

The enclave receives the public key of the client. It creates a shared secret and derives it. It then saves the session key and all the information related to the key and the session to the Redis database. With the newly created shared secret, the enclave adds the following encrypted string to the response payload: *"HELLO YOU =D Love from the enclave ;-)"*. This string will be used as a test to check whether the client has created a correct shared secret in order to decrypt it.

Then it created a JWT without *cid*. The final step is that it returns its public key to the client. The Code 49 generate a public/private key pair in Java.

```
584 public KeyPair genKeyPair() {
585     return kpg.generateKeyPair();
586 }
```

Code 49: Generate a key pair from Java

Code 50 shows the creation of the derived shared secret in Java with the Java private key and the client public key.

```

587 public byte[] genSharedSecret(KeyPair key, byte[]
    OtherPublicKeyBase64) throws ECDHEException {
588     byte[] secret = null;
589     try {
590         KeyFactory kf = KeyFactory.getInstance("EC");
591         X509EncodedKeySpec pkSpec = new
X509EncodedKeySpec(OtherPublicKeyBase64);
592         PublicKey otherPublicKey =
kf.generatePublic(pkSpec);
593         // Key agreement for ECDH
594         KeyAgreement ka =
KeyAgreement.getInstance("ECDH");
595         ka.init(key.getPrivate());
596         ka.doPhase(otherPublicKey, true);
597         byte[] sharedSecret = ka.generateSecret();
598         // derive a key
599         MessageDigest hash =
MessageDigest.getInstance("SHA-256");
600         hash.update(sharedSecret);
601         secret = hash.digest();
602     }
603     catch (NoSuchAlgorithmException |
InvalidKeySpecException | InvalidKeyException ex) {
604         throw new ECDHEException(ex.getMessage());
605     }
606     return secret;
607 }

```

Code 50: Generate shared secret in Java**Step 7**

The request is intercepted by the Zuul post-filter. It does nothing and return the query.

Step 8

The Code 51 shows an example of payload is received and intercepted by Axios.

```

608 Response status 200:
609 {
610     "header": {"kid": B64Url(kid), "alg": "dir"},
611     "ciphertext": "Ciphered(B64Url(HELLO YOU =D Love from the
enclave ;-)))",
612     "iv": "B64Url(iv)"
613     "tag": "B64Url(tag)"
614     "publickey":
"3059301306072A8648CE3D020106082A8648CE3D03010703420004E560F4DD90
249D8A3CFE2C545791D1344D8870D486B481E270D2CBC7C420761EF7CC3F88160
2B8C5E941CFD9E5C8B4C6AA238B852A2D662539100B2E112E16E5"
615 }
616 }

```

Code 51: Example of ciphered payload returned by the enclave

A JWT comes along in the *Authorization* header that contains the *kid*. Now that we have both the client public key and the enclave public key, the process of the creation of the shared secret on the client side can start. This process is much more complicated than on the enclave part and will be detailed here.

Code 52 shows how are generated the public/private and shared secrets on the client side using pure JavaScript and the Web Crypto API.

```

617 function requestNewKey(){
618     return new Promise(async (resolve, reject) => {
619         let alice; // frontend key pair
620         let hello; // enclave welcome message
621         let myIV; // enclave IV
622         let derivedKey; // derived shared secret
623         let kid; // id of the session in the enclave
624         try {
625             // first we generate a new keypair for the client
626             alice = await crypto.generateECDHKeyPair();
627             // Export client public key
628             let publicKey = await
crypto.exportECDHPublicKey(alice.publicKey);
629             // we send the public key to the enclave in Base64Url
630             let res = await axios.post
(`${baseUrl}/1.0/keys/exchange`, { publickey:
utils.arrayBufferToBase64String(publicKey)});
631
632             let enclavePublicKey =
utils.base64StringToArrayBuffer(res.data.publickey);
633             hello = res.data.ciphertext; // enclave message
634             myIV = res.data.iv; // iv BigInteger formatted in base64
635             kid = res.data.header.kid; // key id encoded in base64
636             // we then need to import the enclave public key
637             let epubkey = await
crypto.importPublicKey(enclavePublicKey);
638             // They key is imported and now we use it to generate
the derived shared secret with the frontend private key
639             derivedKey = await
crypto.genDerivedSecret(alice.privateKey, epubkey);
640             // we decrypt the message in the response body with our
newly created shared secret
641             let message = await crypto.decrypt(
642                 derivedKey,
643                 utils.base64StringToArrayBuffer(myIV),
644                 utils.base64StringToArrayBuffer(hello)
645             );
646             if (utils.arrayBufferToString(message) == 'HELLO YOU =D
Love from the enclave ;-){
647                 let secret = await
crypto.exportSecretKey(derivedKey);
648                 resolve(utils.arrayBufferToBase64String(secret));
649             }
650             else
651                 reject('Failed to decrypt enclave message with the
generated shared key');
652         }
653         catch (err){
654             reject(err);
655         }
656     }

```

Code 52: Creation of a shared secret on the client side

As we can see in the code, the first step is to create a key pair on the client. Each key to be used outside the Web Crypto API must be exported. Similarly, if a key comes from outside, it

must be imported. Once the key exchange is done with the enclave, we create our shared secrets and then derive it. To test the encryption function, the "hello" message must be sent encrypted from the enclave and must be decrypted correctly. Once decrypted, the session key and the JWT are stored in the browser's *localStorage*. The *crypto* class is a wrapper that contains the Web Crypto API code.

Code 53 is used to create and derive a key.

```

657 function genDerivedSecret(privateKey, publicKey){
658     return new Promise(async (resolve, reject) => {
659         try {
660             let keydata = await window.crypto.subtle.deriveKey({
661                 name: 'ecdh',
662                 namedCurve: 'P-256',
663                 public: publicKey,
664                 //hash: 'SHA-256', // does not work, bug ?
665             },
666                 privateKey,
667                 { name: 'AES-GCM',
668                     length: 256},
669                 true, // extractable
670                 ['encrypt', 'decrypt']); // use to encrypt and
        decrypt data
671             let exportedKey = await
        window.crypto.subtle.exportKey('raw', keydata);
672             let exportedDigestedKey = await
        crypto.subtle.digest('SHA-256', exportedKey);
673             let importedDigestedKey = await
        window.crypto.subtle.importKey(
674                 'raw', // raw is full binary
675                 exportedDigestedKey,
676                 { name: 'AES-GCM',
677                     namedCurve: 'P-256'},
678                 true, // extractable
679                 ['encrypt', 'decrypt']); // use to encrypt and
        decrypt data
680             resolve(importedDigestedKey);
681         }
682         catch (err){
683             reject(err);
684         }
685     });
686 }

```

Code 53: Shared secret creation and derivation with web crypto API

The rest of the Web Crypto API code to export/import keys, encrypt/decrypt are pretty straightforward and comments in the code help to understand the details.

By default, there is no support for BigInteger in Web Crypto API. In order to support BigInteger, the BigNumber.js pure JavaScript library was used. Particular attention must be paid to the choice of the library. Indeed, the most used library to manipulate large numbers in JavaScript is BigInteger.js. However, during the development, this library generated incorrect big numbers without any warning, which greatly delayed development.

3.9.3 Authentication

Once the session keys have been established, it becomes then possible to exchange messages following the protocol established in the theoretical part. We can now communicate with the enclave, but it is necessary to authenticate because for the moment, access to resources is subject to authorization.

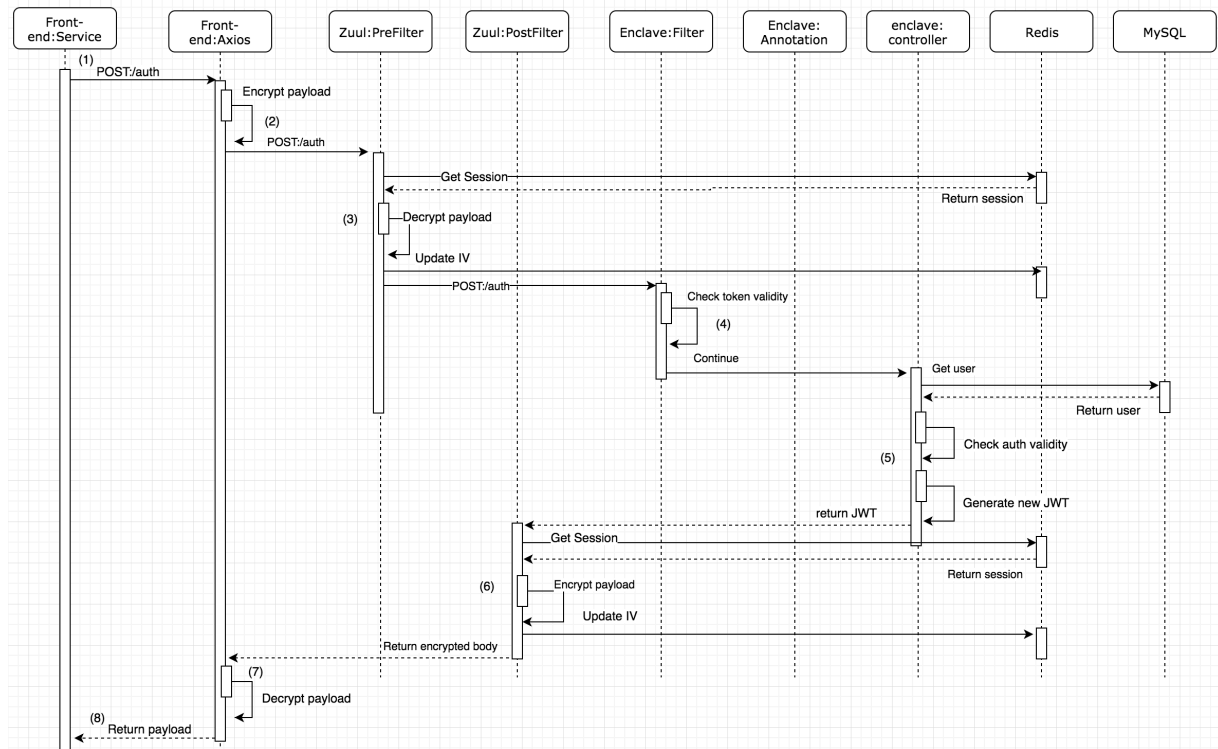


Figure 28: Authentication sequence

The UML sequence diagram in Figure 28 shows the steps required to authenticate on the enclave. We assume here that the session key was already established.

Step 1

The service layer creates a *POST:/1.0/auth* request with the password and email address in the request body.

Step 2

The request is intercepted by Axios, which will retrieve the session key from the *localStorage* and then encrypt the request body using the protocol. Code 54 allows Axios to intercept requests and encrypt them.


```

687   axios.interceptors.request.use(async config => {
688     // do we encrypt at all ?
689     if (!consts.doCrypto) return config;
690     // we won't encrypt this url
691     if (config.url.endsWith(`/1.0/keys/exchange`)) return config;
692
693     // everything below will be encrypted
694     let key = auth.getKeys();
695     let token = auth.getToken();
696     // Set the token if it exists
697     if (token !== null) config.headers['Authorization'] =
token.token;
698     //Replace the body content by the ciphered body content
699     let secret = utils.base64StringToArrayBuffer(key);
700     let iv      =
crypto.incrementIV(utils.base64StringToArrayBuffer(token.iv));
701     try {
702       let iKey = await crypto.importSecretKey(secret);
703       let cipheredData = await crypto.encrypt(iKey, iv,
utils.stringToArrayBuffer(JSON.stringify(config.data)));
704       cipheredData =
utils.arrayBufferToBase64String(cipheredData);
705       config.data = {
706         header: {
707           alg: 'dir',
708           kid: token.header.kid
709         },
710         iv: utils.arrayBufferToBase64String(iv),
711         ciphertext: cipheredData,
712         tag: '1234'
713       };
714       return config;
715     }
716     catch(err){
717       console.log(err);
718     }
719   }, error => {
720     return Promise.reject(error);
721   });

```

Code 54: Intercept HTTP request with Axios

Step 3

The request is intercepted by Zuul. It will at first get the session key in Redis using the *kid* provided by the client. Then it will decrypt the request body, update the *iv* in Redis and send the request to the enclave. The code used to decide whether the request body will be decrypted or not and the actual decryption code is in Code 55.

```

722  @Override
723  public boolean shouldFilter() {
724      RequestContext ctx = getCurrentContext();
725      HttpServletRequest request = ctx.getRequest();
726      String uri = request.getRequestURI().toString();
727      if (request.getMethod().equals("GET") ||
request.getMethod().equals("DELETE") ||
uri.endsWith("/1.0/keys/exchange")) {
728          System.out.println("NO decryption needed");
729          return false;
730      }
731      return true;
732  }
733  @Override
734  public Object run() {
735      try {
736          RequestContext context = getCurrentContext();
737          InputStream in = (InputStream) context.get("requestEntity");
738          if (in == null) in = context.getRequest().getInputStream();
739          // encrypted body
740          String body = StreamUtils.copyToString(in,
Charset.forName("UTF-8"));
741          EncryptedPayload o = new Gson().fromJson(body,
EncryptedPayload.class);
742          String kid = o.getHeader().getKid();
743          String iv = o.getIv();
744          String ciphertext = o.getCiphertext();
745          SessionKeyModel skm = skp.findOne(kid);
746          // decrypt the payload as an array of bytes
747          byte[] ivByteArray = ch.Base64Url2byteArray(iv);
748          byte[] message = aes.decrypt(
749              ch.Base64Url2byteArray(ciphertext),
750              ch.Base64Url2byteArray(skm.getSharedKey()),
751              ivByteArray
752          );
753          // increment the IV and save it in Redis
754          String newIV =
ch.byteArray2Base64Url(aes.nextIV(ivByteArray));
755          skm.setIv(newIV);
756          skp.findOne(kid);
757          // replace the encrypted body by the unencrypted body
758          context.setRequest(new
HttpServletRequestWrapper(getCurrentContext().getRequest()) {
759              @Override
760              public ServletInputStream getInputStream() throws
IOException { return new ServletInputStreamWrapper(message); }
761              @Override
762              public int getLength() {return message.length;}
763              @Override
764              public long getLengthLong() {return message.length;}
765          });
766      }
767      catch (Exception e) {
768          rethrowRuntimeException(e);
769      }
770      return null;
771  }

```

Code 55: Decryption of request payload by Zuul

Step 4

The request is intercepted by the enclave filter. It will check if the token is valid and was issued by the enclave. Once it's done, it will send the request to the controller.

Step 5

The authentication controller will get the tuple in the database corresponding to the values provided by the client. If there is a correspondence, the controller will create a new JWT but this time with the user *cid* in the header. The presence of the *cid* in the header of the JWT means that the user has provided correct credentials and is authenticated. The enclave returns the token in both the response body and the Authorization header.

Step 6

The response body is intercepted by Zuul. It will get the session key in Redis with the *kid* provided by the enclave. It will then increment the *iv* and encrypt the response body with a very similar code used in **step 3**, but instead it will be used for encryption. Once the body is ciphered, the Zuul gateway updates the *iv* in Redis. The ciphered response body is returned to Axios.

Step 7

The response body is intercepted by Axios. It will decrypt the response body with a very similar code from **step 2** and it will save and increments the JWT and *iv* in the *localStorage*. Once done it returns the decrypted response body to the service layer.

Step 8

The service layer gets the message and return the content to the caller.

3.9.4 Message exchange

Once the session key has been defined and the user has authenticated, it is then possible to exchange messages with the enclave in a secure way. The following diagram shows the steps for message exchange between a client and the enclave:

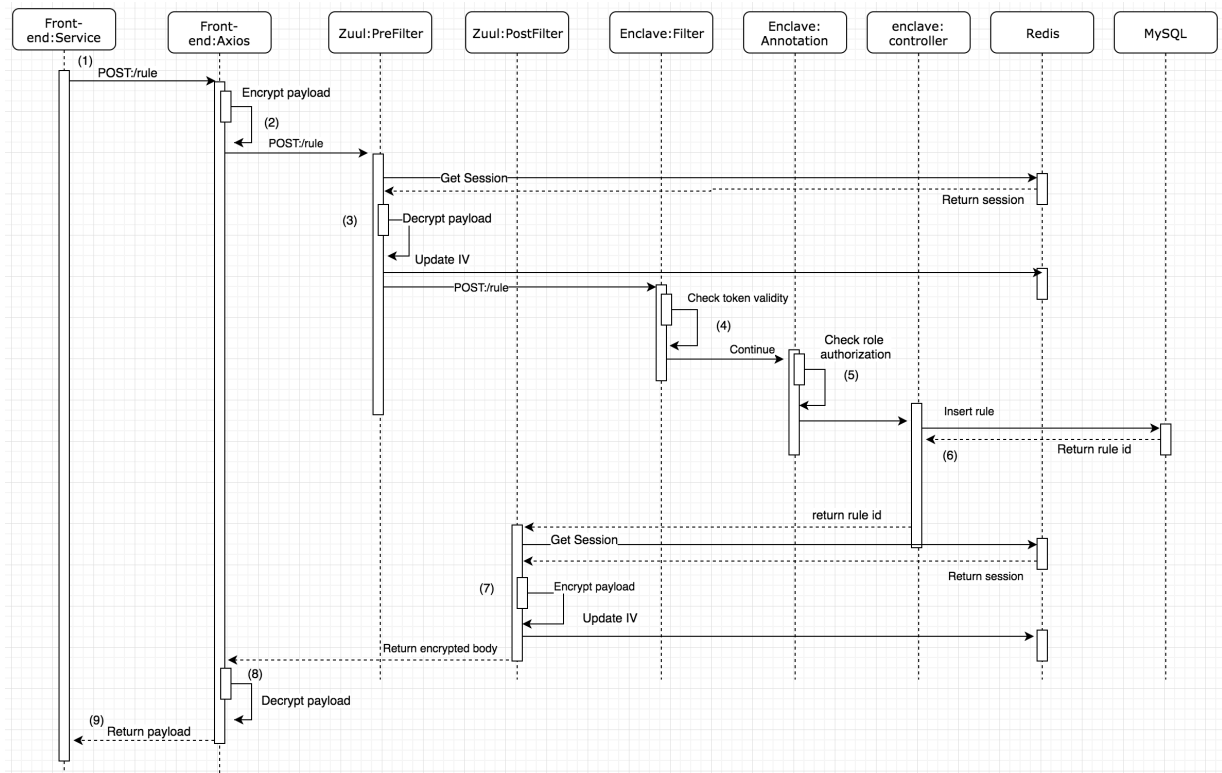


Figure 29: Message exchange sequence

The UML sequence in Figure 29 shows the sequence of message exchange between an authenticated client and the enclave where the client creates a new rule. Here the sequence is very similar to the login. The difference comes at **step 5** where the enclave performs an authorization control with an annotation to access the resource. It will extract the JWT provided by the client, then check if its role in the JWT matches the role defined in the route annotation.

3.10 Software development organization

3.10.1 Source code version control

The whole project was developed on a local machine, and each version of the code was committed on a Github private repository. The following private repositories have been created and are accessible by all project stakeholders:

- <https://github.com/grosjeanma/TM-enclave> (enclave, Zuul gateway & email sender)
- <https://github.com/grosjeanma/TM-frontend>
- <https://github.com/grosjeanma/TM-documentation>
- <https://github.com/grosjeanma/TM-client>
- <https://github.com/grosjeanma/TM-benchmark-crypto>

3.10.2 Iterative Development

An iterative development approach was chosen to develop this software artifact. The Design Science Research methodology tells us that at the beginning of an original research project, the theoretical and practical issues of the project must be presented in the theoretical part. Then, based on the theoretical part, it is necessary to quickly create a prototype that meets the needs in order to evaluate it and to arrive to a final prototype that meets the initial challenges, requirements and advances science.

In this project, we covered the initial research project on the middleware and the technologies it uses. Then the iFlux project was presented as well as the different cryptographic methods to ensure point-to-point security. Based on these elements, a prototype was created and sent for evaluation to assess the relevance of the artifact created. Based on the remarks, the prototype was refined.

3.10.3 Testing

Introduction

It is important to implement some automated tests to validate the quality of the code that has been developed. They are important because they validate the proper functioning of the code on different platforms as well as with different versions. Automatic tests allow us to get a quick overview of the current quality of the code and whether it is necessary to fix problems before developing new features. We will test bits of code automatically with JUnit. We will not test the architecture as a whole with all components working together because of its complexity. Performance tests will not be performed here.

Gateway

For the gateway, we will run one unit test to test the key generation and symmetric cryptography function. We will use maven in order to run the JUnit:

```
772 mvn test
```

Code 56: Gateway automated tests

There should be no error in the console output.

Email sender

The email sender component, or action target, implements actuators that are called by the enclave. This component is developed with NodeJS and exposes a REST API. There are frameworks to test REST APIs such as Mocha and which allow to automate testing. However, in our case, we will just use a simple CURL request to test sending an email with the command in Code 57.

```
773 curl --header "Content-Type: application/json" \  
774      --request POST \  
775      --data \  
       '{"sender": "marcel.grosjean@unine.ch", "dest": "marcel.grosjean@uni  
       ne.ch", "subject": "test", "body": "test"}' \  
776      http://localhost:7010/emails
```

Code 57: Email sender CURL test

An email should have been sent.

Enclave

For the enclave, we will run two unit tests to test the key generation, symmetric cryptography function and database connectivity. We will use maven in order to run the JUnit test.

```
777 mvn test
```

Code 58: Enclave automated tests

There should be no error in the console output.

Front-end

Automated testing can be done in Single Page Application (SPA) using the WebDriver.io application, for example. It executes SPA pages and put values in the code of the pages generated in order to test the user interface. In our case, manual tests were performed because the interface is rather small and therefore easy to test manually.

3.10.4 Deployment

Introduction

The infrastructure developed during this project is complex. We have developed a multi-tier software architecture. Indeed, we have a front end that communicates with an enclave. The data is intercepted by the gateway, and actions are taken on remote targets. Two databases, Redis and MySQL, are used to manage persistent data. We can count the following 7 third parties: front end, Zuul, enclave, action targets (1 only in our case), the client, Redis and MySQL.

It is therefore necessary to document in detail the deployment procedures in order to be able to run the architecture. A precise deployment procedure allows a team of developers to take over the project more easily in order to apply updates and continue to add functionalities. In this section, we will cover two ways of deploying software: local deployment and in Docker containers.

Local Deployment

By local deployment we mean the following: the local machine, a remote physical server, a remote virtual server (VPS) or a platform on the cloud (Amazon EC2, ...). Local deployment is the oldest and most common solution for deploying an application. Indeed, it is necessary to install all the dependencies, compile and execute the binaries in order to have a version that runs locally. The problem is that it is necessary to install in advance all the dependencies, libraries and external components in order to make it work. Versions management of all the dependencies can very quickly become a problem because depending on the OS and version of the app. Local deployment can become very complex. In addition, it is necessary to monitor the

state of the machine in case of a failure or insufficient resources. The system administrator is responsible for the scalability and updates (both hardware and software).

Although the deployment of a local infrastructure can be complex and labor-intensive, it is still the simplest and most common way to deploy an application.

A procedure for installation of the dependencies, compilation, execution for testing and execution as a service has been done and is available in appendix for the following components: front end, Zuul gateway, enclave and email sender. By carefully following the procedures, it will be possible to deploy the infrastructure locally for testing and production purposes.

Docker

Docker allows developers to deploy and execute code efficiently. A container wraps everything an application needs to run. This includes the operating system, application code, system tools, libraries and interactions with other containers, databases or external API. Docker containers are built from Docker images. They are lightweight, portable and allow developers to efficiently deploy and run distributed applications. In addition, they allow an application to be packaged and moved easily, increasing the simplicity of an infrastructure. Containers are less isolated from each other than virtual machines and inter-container communication and access to the code in a container is easier [35].

The difference between Docker and our local deployment procedure is that it is no longer necessary to install all the components ourselves. By just specifying which components and libraries we need to run, Docker takes care of downloading and installing all the dependencies. This is very useful for sharing our code for third-party developers but also to deploy our application. Indeed, once we have created our Docker images and everything works locally, it is just necessary to send the images to the cloud so that it can build the architecture for production on its own. Admittedly, there is a time cost for the installation of the various containers and their proper functioning. But once everything is working, deployment is incredibly easy. In a few minutes it would therefore be possible to deploy our project locally or in the cloud. Not to mention that the scalability is also made easier thanks to the containers.

In this project, in addition to the local deployment procedure, the procedure for creating Docker images and the execution of containers was documented. The advantage here is that the project will be incorporated into more global academic research and having an easier means of deployment allows researchers to easily take over the project and adapt it to their research. The following containers were created to run the project: One for the front end, one for the enclave, one for the gateway, one for the action target, one for MySQL and one for Redis. Simply follow the procedure described in the appendix in the READMEs to create the Docker containers and launch them.

3.11 Conclusion

In this section, we have seen the choices made both in terms of architectural design and technical choices. Our risk analysis indicates that the difficulties are mainly at the technical implementation level and that all the technologies used for development exist or are under development. The risks we had identified can all be mitigated because for each of them, there is an alternative.

The overall architecture is a multi-tier architecture. We do indeed have a front end which is the main artifact of this project. To run the front end, it was necessary to develop all the infrastructure around it in order to simulate the final environment in which it will be executed. Vue.js was chosen for the development of the interface for the simple reason that it was the most popular in 2018.

Encryption is necessary, so we decided to make it as transparent as possible to the web developer. It was therefore decided to create an encryption middleware to intercept the requests and encrypt/decrypt them in a transparent way. The advantage is that the web developer can focus on developing features rather than dealing with cryptographic functions. The new Web Crypto API native implementation has been used for client-side encryption. The data are intercepted by the Axios middleware and ciphered before being sent to the enclave. The request is then intercepted by the Zuul reverse proxy in order to be deciphered before sending it to the enclave. Cryptography is thus transparent to the eyes of the front end and the enclave. The enclave is responsible for authentication and authorization.

In order to implement the ECA paradigm in our enclave, we took the iFlux project developed by the HEIG-VD which exposes the concepts of event sources, action targets and rules. We used these concepts to create our own REST API with Spring Boot. After implementing our own version of these concepts, we were able to process data from Things, infer on this data according to rules and trigger actions with an action target. To be able to infer on the data, we have created a rule engine that dynamically generate JavaScript code that will be evaluated by the enclave rule engine. The action targets are actuators that can be either software or hardware. We have developed an action target that sends emails and exposes a REST API. It can be accessed by the enclave.

In order to facilitate the deployment of the entire project, a deployment procedure for testing, another procedure for production and a procedure for creating a Docker image has been created.

We have seen in this section that the project was very complex both in terms of the concepts to be implemented and the technologies that haven been used. These are risks and fortunately each of these risks had alternatives if a problem was met. The front end that has been developed meets the defined use cases. The cryptography middleware implements the protocol defined and operates transparently. The enclave developed uses iFlux concepts to implement the ECA paradigm. The rules engine works and allows actions to be triggered according to rules and events. And finally, the action target executes the requested actions. From a purely functional point of view, this project is a success because it meets both technical and conceptual expectations. It is also easily portable, which makes it easier to take over within the context of the current Softeng Group project.

3.12 Improvements & Future Works

There are many areas for improvement in this project. The main goal is to develop a web application. For the moment the application responds to all the use cases defined but it would be interesting to discover other use cases. In a second step, the message exchange protocol could be improved. Indeed, the original protocol proposed at the beginning of the project is an adaptation based on the ECDHOC [36] protocol. However, it was not retained during development because it was considered too difficult to implement and a lighter version was developed instead. One possible improvement would be to modify the current protocol to correspond to the one proposed by EDHOC.

Improvements can also be applied to the enclave and especially on the rule engine. Indeed, when a client sends an event, the enclave processes it and sends it to the rule engine that will infer on it and then trigger an action if necessary. All these steps are made synchronously. If a very large number of events occur concurrently, this could create a problem because the entire process would take too long. One possible improvement would be for the enclave to send the event in a queue. The rule engine would run in another process and consume the events in this queue. Thus, the enclave would only be concerned with controlling that the event is in the right format and would have more resources to handle many concurrent requests. Other modifications on the enclave, gateway or target actions are possible but are not considered because they will not be included in the project led by the Softeng group.

4

Practical testing and evaluation

4.1 Introduction	102
4.2 RIOT scenario	103
4.2.1 Introduction	103
4.2.2 Login	104
4.2.3 Verify session key	105
4.2.4 Users	107
4.2.5 Clients	107
4.2.6 Urls (action targets urls)	108
4.2.7 Event type	109
4.2.8 Action type	110
4.2.9 Rules	111
4.2.10 Graph explorer	114
4.3 Client scenario	115
4.3.1 Introduction	115
4.3.2 Client	116
4.3.3 Emails sent	118
4.3.4 Events list	119
4.3.5 Action message	120
4.4 Conclusion	120

4.1 Introduction

In this section we will briefly present the web application developed in this project with a practical case study. This case study will cover most of the use cases defined in the project and we will test the configuration of the rule engine with a client that we have developed specifically for testing. We will then test our enclave configured by our web application with real life data from our client.

Throughout the theoretical part, we talked about how this project could improve our lives and in particular how our quality of life could be improved if we made our cities smart. However, for the tests we will take a different point of view and use our project for spying purposes. We're going to spy on people with a camera and a microphone.

We will measure two things. The first is what people say with the help of a microphone. We have created an application that captures the sound of microphones and processes language in order to understand what people are saying. The second thing is the facial emotion that people have when they talk. This emotion is captured with a camera. For example, we can recognize through the microphone, if a person says suspicious things about another person and what their feeling is about that person.

For example, we want to install our client on all smartphones of the citizen of a country to find out what people think about the president. If the citizen quotes the president's name during a discussion and have an angry face while talking, then action could be taken.

Another practical application would be to install our client at airport camera when foreigners arrive in custom areas. We could also hide microphones to listen to what they say. If a person in the crowd has suspicious comments and a nasty look, then the custom officer would be informed before that person arrives at the desk.

We will therefore feed our rule engine with data from the microphone and camera. Once the rule engine has identified an action to execute, we will fire the associated action targets. In our practical case, we will perform two actions. The first is sending a logging email just saying that an action has been taken and the second is still an email but with more detailed information such as a picture of the suspect individual with what he or she said.

4.2 RIOT scenario

4.2.1 Introduction

In order to configure the enclave with RIOT, we will execute the following scenario:

- Login into the application and configure the user account.
- Verify the session key used by the interface.
- Check the details of the connected user.
- Create two clients. One as *smartphone_marcel* and the second as *email_server*.
- Create two URLs. One for logging emails and the other for an email with a photo. They are both associated with the *email_server* client.
- Create the event type *Photo event* that will be used to monitor the angeriness of a person. It takes as parameters the *photo* (string), the *angry* (number), and the *talking* (string).
- Create two action types. One for sending an email with the following properties: *sender* (string), *subject* (string), *body* (string) and *dest* (string). The second action type is similar to the first one but with the following properties added: *photo* (string) and *angry* (number).
- Create a rule that trigger an action for the event type *Photo event* and the client *smartphone_marcel* where the action will be triggered if the user is angry (angeriness > 0.5) and the recorded text by the microphone contains “*Trumps*” or “*Obama*”. If the rule is triggered, then do the following actions:
 - Send a simple logging email.
 - Send a more sophisticated email with the photo of the person, a formatted text and the angeriness rounded at the tenth superior.
- Check the result of all the actions in the graph explorer

4.2.2 Login

The login page in Figure 30 is the only page accessible without being logged. The button at the top left allows to show/hide the navigation menu.

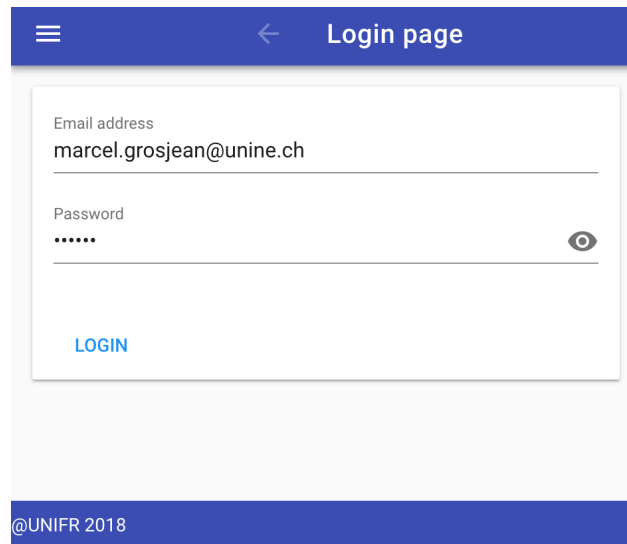


Figure 30: Login page

Once the user is logged in, he will be taken to the home page as in Figure 31. It displays all the available pages. The *Navigation* section is accessible by normal users while the *Administration* section is accessible by administrators only.

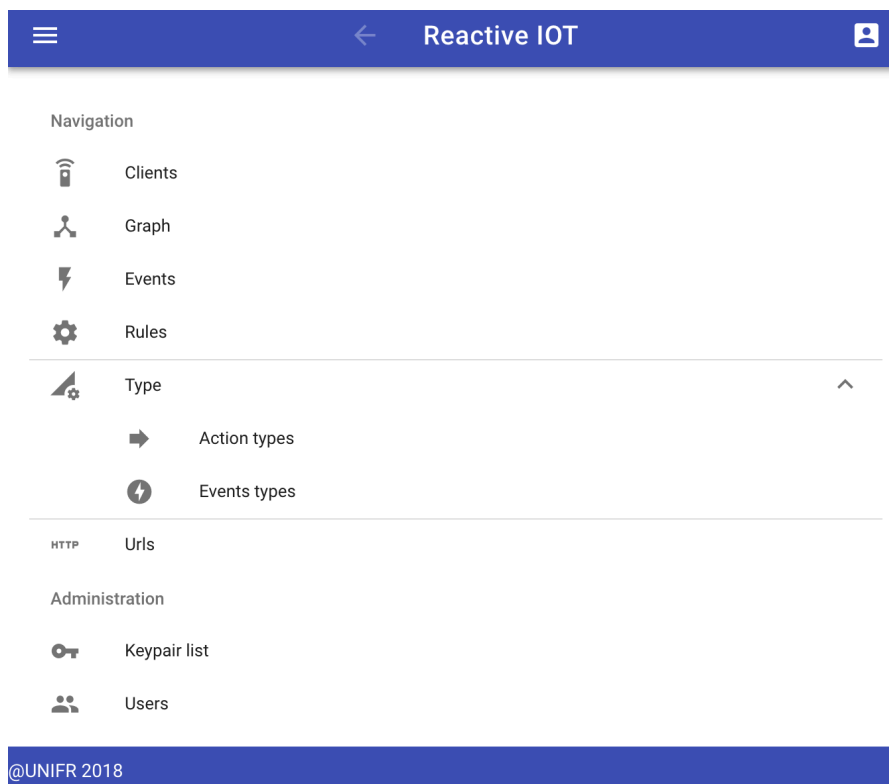
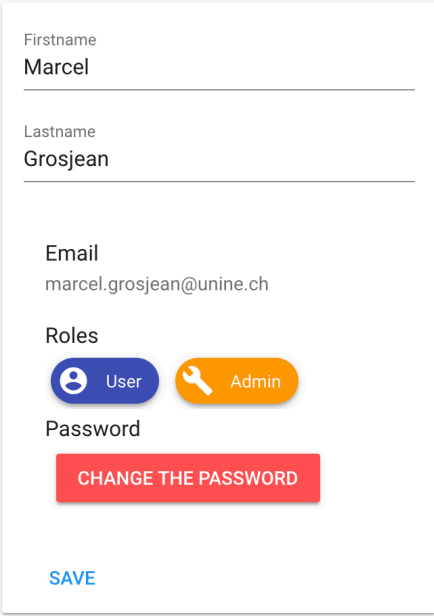


Figure 31: Home page

The button at the top right allows the user to either log out or access the user's configuration page as in Figure 32.



Firstname
Marcel

Lastname
Grosjean

Email
marcel.grosjean@unine.ch

Roles
User Admin

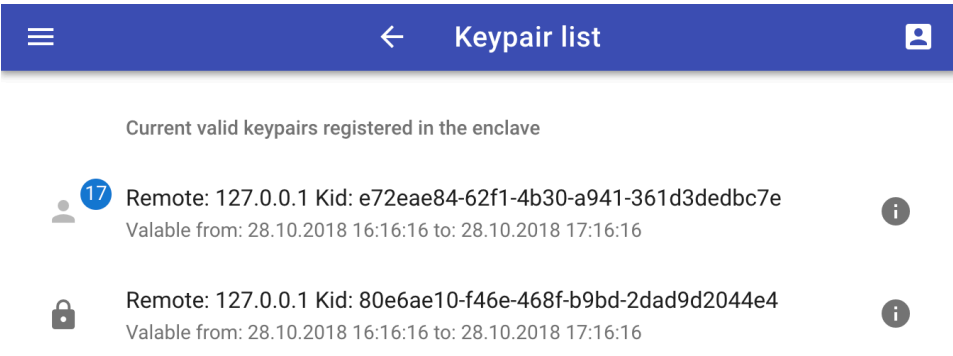
Password
CHANGE THE PASSWORD

SAVE

Figure 32: User configuration page

4.2.3 Verify session key

The session key page shows which session keys are generated and currently valid. The number 17 in Figure 33 next to the user icon means that this session has been assigned to the user with id 17.



Menu icon ← Keypair list User icon

Current valid keypairs registered in the enclave





 17	Remote: 127.0.0.1 Kid: e72eae84-62f1-4b30-a941-361d3dedbc7e Valable from: 28.10.2018 16:16:16 to: 28.10.2018 17:16:16	
	Remote: 127.0.0.1 Kid: 80e6ae10-f46e-468f-b9bd-2dad9d2044e4 Valable from: 28.10.2018 16:16:16 to: 28.10.2018 17:16:16	

Figure 33: Keys list page

The session key detail shows all the details associated with the session. These are the data present in Redis. It is possible at any time to revoke a session key. Figure 34 show the session page.

Keypair details

KID:
 Base64Url: ZTcyZWFIODQtNjJmMS00YjMwLWE5NDEtMzYxZDNkZWRIYzdl
 String: e72eae84-62f1-4b30-a941-361d3dedbc7e

Enclave Public Key:
 Base64Url: MFkwEwYHKOZlZj0CAQYIKoZlZj0DAQcDQgAEVeHwR01Z1lagKKaxf0dqghjD7N4clu
 eD06kKuEmU-4hLH50A
 HEX: 3059301306072a8648ce3d020106082a8648ce3d0301070342000455e1f0474d59

Enclave Private Key:
 Base64Url: MEECAQAwEwYHKOZlZj0CAQYIKoZlZj0DAQcEJzAlAgEBBCB7qIV068QW4t-
 fSdr5DGBZQzPchFHctjhNcDKsobc9_w
 HEX: 3041020100301306072a8648ce3d020106082a8648ce3d03010704273025020101

Derived shared secret:
 Base64Url: BilYBzW3JcZL0l6Hv9K6Tv1ZsBzFOUyyIMfrAAxxxJE
 HEX: 0629580735b725c64bd25e87bfd2ba4efd59b01cc5394cb220c7eb000c71c491

External public key:
 Base64Url: MFkwEwYHKOZlZj0CAQYIKoZlZj0DAQcDQgAEjkhV7R6YjXmZ835sRCVJoPLMLgrTe
 AvF6GRal84LjCjA
 HEX: 3059301306072a8648ce3d020106082a8648ce3d030107034200048e4855ed1e98

IV:
 Base64Url: LTlZOTA1NDIxOTY4MTkzODM0MTgzNDE2ODk1MTZH
 BigInteger: -2390542196819383418341689516G

Userid:
 17

Remote address:
 127.0.0.1

Created_at:
 28.10.2018 16:16:16

Expire_at:
 28.10.2018 17:16:16

REVOKE THIS KEYPAIR

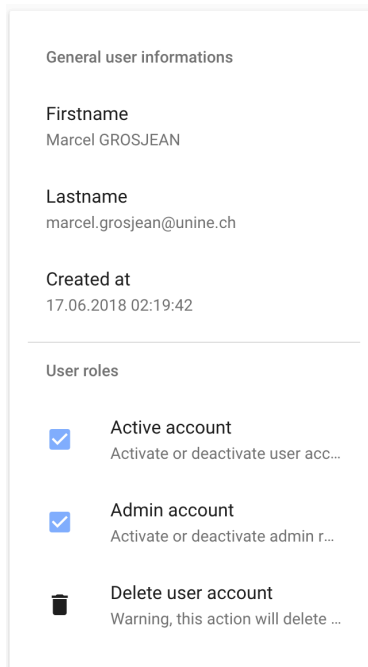
CLOSE

@UNIFR 2018

Figure 34: Session key details page

4.2.4 Users

An administrator can list and filter users. He can then define if this user is active and create or delete a user as in Figure 35.



General user informations

Firstname
Marcel GROSJEAN

Lastname
marcel.grosjean@unine.ch

Created at
17.06.2018 02:19:42

User roles


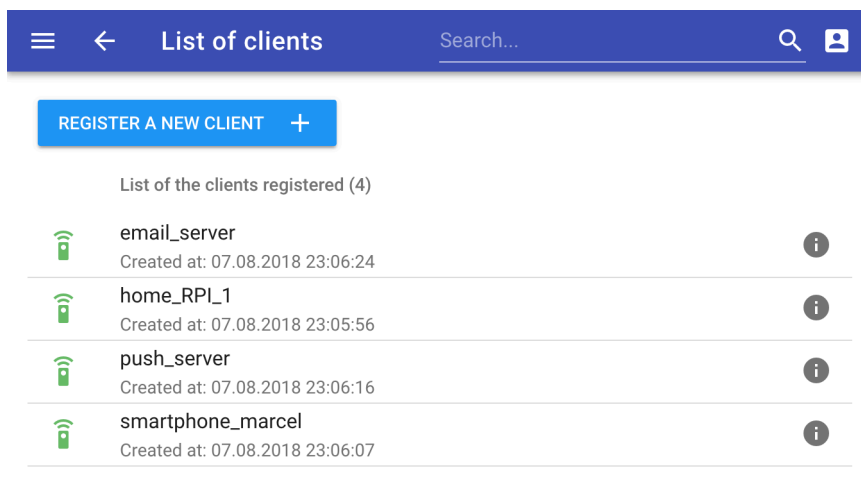
- ☒ **Active account**
Activate or deactivate user acc...
- ☒ **Admin account**
Activate or deactivate admin r...
-  **Delete user account**
Warning, this action will delete ...

Figure 35: User details page for the administrator

4.2.5 Clients

The clients page shows all the clients. The user can create or filter all the clients. The green icons mean that the clients is active. Red icons mean inactive client. Figure 36 illustrates this.














List of clients			Search...		
REGISTER A NEW CLIENT 					
List of the clients registered (4)					
	email_server		Created at: 07.08.2018 23:06:24		
	home_RPI_1		Created at: 07.08.2018 23:05:56		
	push_server		Created at: 07.08.2018 23:06:16		
	smartphone_marcel		Created at: 07.08.2018 23:06:07		

Figure 36: Clients list page

Figure 37 shows the client details page. A client has a name and a public key. The public key is not used here. A client can have several URLs and they can be added from here. The user can define here if the client is admin or active and can delete the client if needed. From here, the user can see all the action messages for this client.

Client name
smartphone_marcel

Client public key
abc

NEW URL +

List of Urls (0)

☒ Admin client

☒ Active client

Action message
See all the actions message fo...

Delete this client
Warning this action will delete ...

UPDATE

Figure 37: Clients details page

4.2.6 Urls (action targets urls)

Figure 38 shows the list of the URLs. The user can filter or add a URL from this page.

≡ < List of Urls Search... 🔍 👤

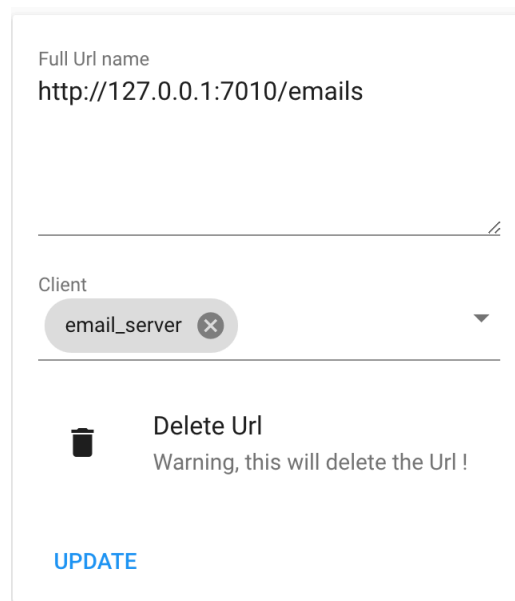
NEW URL +

List of Urls (4)

http://127.0.0.1:7003/push	push_server	
http://127.0.0.1:7010/emails	email_server	
http://127.0.0.1:7010/emails/photos	email_server	
http://127.0.0.1:7010/emails/putin	email_server	

Figure 38: Urls list page

Figure 39 shows the details of a url. A url is associated with a client.



Full Url name
http://127.0.0.1:7010/emails

Client
email_server

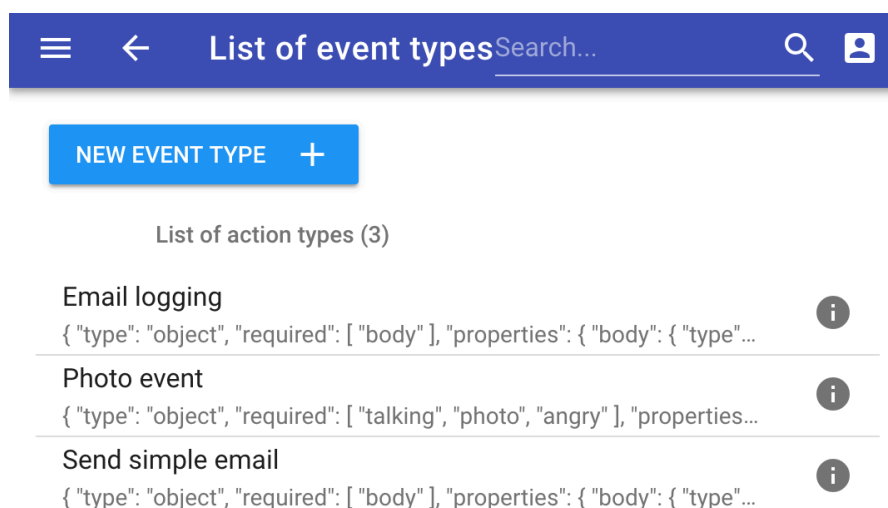
Delete Url
Warning, this will delete the Url !

UPDATE

Figure 39: Url details page

4.2.7 Event type

Figure 40 lists all the event types. They can be filtered or added here.



≡ ← List of event types Search... 🔍 👤

NEW EVENT TYPE +

List of action types (3)

Email logging	{ "type": "object", "required": ["body"], "properties": { "body": { "type"...	i
Photo event	{ "type": "object", "required": ["talking", "photo", "angry"], "properties...	i
Send simple email	{ "type": "object", "required": ["body"], "properties": { "body": { "type"...	i

Figure 40: Event types list page

Figure 41 shows the details of an event type. Here the event type *Photo event* has 3 mandatory properties. The *photo* (string), *angry* (number) and *talking* (string). Parameters can be freely added/removed and can be set as required or not. The available data types are string, number and integer.

The screenshot shows the 'Photo event' details page. At the top, there's a blue header with a menu icon, a back arrow, the title 'Photo event', and a user profile icon. Below the header, the 'Event type name' is 'Photo event'. Under 'New scheme:', there's a table with columns 'Type name' and 'Type'. The 'Type' column has a dropdown menu currently set to 'integer', a checkmark icon, and a plus icon. Below this, the 'Scheme:' section lists three properties: 'photo': 'string' * required, 'angry': 'number' * required, and 'talking': 'string' * required. Each property has a red 'X' icon to its right. At the bottom, there's a trash can icon with the text 'Delete this event type' and a warning: 'Warning, this action will delete the event type !'. A blue 'UPDATE' button is at the very bottom.

Figure 41: Event type details page

4.2.8 Action type

Figure 42 lists all the action types. They can be filtered or added here.

The screenshot shows the 'List of action types' page. At the top, there's a blue header with a menu icon, a back arrow, the title 'List of action types', a search bar with the placeholder 'Search...', a magnifying glass icon, and a user profile icon. Below the header, there's a blue button with the text 'NEW ACTION TYPE' and a plus icon. Underneath, it says 'List of action types (2)'. There are two action types listed: 'Send email with photo' and 'Send simple email'. Each entry shows a JSON snippet of its schema and has an information icon (i) to its right.

Figure 42: Action type list page

Figure 43 shows the details of an action type that aims to send an email with multiple parameters. Here the action type *Send email with photo* has 6 mandatory properties. The *photo* (string), *angry* (number), *talking* (string), *sender* (string), *subject* (string), *body* (string) and *dest* (string). Parameters can be freely added/removed and can be set as required or not. The available data types are string, number and integer.

The screenshot shows a web interface for editing an action type. At the top, it says 'Action types' and 'Send email with photo'. Below this is a 'New scheme:' section with a 'Type name' input field, a 'Type' dropdown menu set to 'integer', a checked checkbox, and a '+' button. Underneath is a 'Scheme:' section with a table of properties:

Property	Value
"sender": "string" * required	✗
"subject": "string" * required	✗
"photo": "string" * required	✗
"angry": "number" * required	✗
"body": "string" * required	✗
"dest": "string" * required	✗

Below the table is a 'Delete this action type' button with a warning message: 'Warning, this will delete the action type !'. At the bottom is an 'UPDATE' button.

Figure 43: Action type details page

4.2.9 Rules

Figure 44 lists all the rules of the enclave. They can be added or filtered from here. An active rule is green while an inactive one is gray.

The screenshot shows a web interface for listing rules. At the top is a blue header bar with a menu icon, a back arrow, the title 'List of rules', a search bar, and a magnifying glass icon. Below the header is a 'NEW RULE +' button. Underneath is a section titled 'List of the rules (2)' containing two rules:

Rule Name	Status	Info Icon
OBAMA TRUMP photo	Active (Green)	Info (i)
Vladimir Putin rule	Inactive (Gray)	Info (i)

Figure 44: Rules list page

The rules window in Figure 45 is divided into two parts. The left part which makes it possible to define a name for a rule which, once added, is associated with only one type of event, in our case the *Photo event*. This rule can be used by one or more clients. Here, our rule exists for the *smartphone_marcel* client who will send a *Photo event* type event. A rule can be disabled, in which case it will not be taken into account by the rule engine.

The right part is used to define which conditions will allow the rule to be triggered and which actions to perform if the rule is triggered. At the top right is the list of properties associated with the event type that will be used to build the function. The function must be a valid JavaScript condition, and the variables must be in the event type properties. The condition says that if the *angriness* is greater than 0.5 and the captured text contains "*Trump*" or "*Obama*" then the associated actions are triggered.

The lower right part is dedicated to the actions of this rule. We can either add an action or modify an action. Here we have two actions. The first one is to send a simple email and the second one is to send an email with a photo and the *angriness*.

The screenshot shows the 'OBAMA TRUMP photo' rule configuration page. The interface is as follows:

- Header:** A blue bar with a menu icon, a back arrow, the title 'OBAMA TRUMP photo', and a user profile icon.
- Left Panel (Configuration):**
 - Rule name:** OBAMA TRUMP photo
 - Event type:** Photo event (with a dropdown arrow)
 - Clients:** smartphone_marcel (with a dropdown arrow)
 - Active rule:** Indicated by a checked checkbox. Description: 'This action will activate/deactivate a rule'.
 - Delete this rule:** Indicated by a trash icon. Description: 'Warning, this action will delete the rule !'.
 - UPDATE:** A blue button at the bottom left.
- Right Panel (Logic and Actions):**
 - Event properties:**
 - photo:string *required
 - angry:number *required
 - talking:string *required
 - Function:**

```
angry > 0.5 && (talking.toLowerCase().contains('trump') || talking.toLowerCase().contains('obama'))
```
 - Actions (2):**
 - Send simple email (with an info icon)
 - Send email with photo (with an info icon)
 - Buttons:** A blue 'NEW ACTION +' button is located above the actions list.

Figure 45: Rule details page

Figure 46 shows the details for the *Send email with photo* action. It is divided into two parts.

The left part defines what this action is made of. An action is composed of an action type and an URL. A reminder of the event type and the action type is displayed to help build the action.

The right part allows you to define the structure of the action. An action property can have several types: *string*, *integer* and *number*. If the action is a string, then we can return either literally a string or a string with variables from events. The property *sender* is a string. The *photo* property is only the content of the *photo* property of the event type. The *body* property is a mixture between string and the content of the variable `{{talking}}` will be replaced by the content of the *talking* variable in the event. If the property is an integer or number, it is possible to either return directly the content or perform data transformation. The *angry* property is a JavaScript expression that rounds up the content of the *angry* variable that comes from the event.

Edit action

Action type: Send email with photo

Urls: http://127.0.0.1:7010/emails/photos

Event properties:

- photo:string *required
- angry:number *required
- talking:string *required

Action properties:

- sender:string *required
- subject:string *required
- photo:string *required
- angry:number *required
- body:string *required
- dest:string *required

Delete this action
Warning, this action will delete the action !

Action properties:

- sender:string
marcel.grosjean@unine.ch
- subject:string
Mean talking about Trump or Obama
- photo:string
{{photo}}
- angry:number
Math.round(angry * 10) / 10
- body:string
A citizen said mean things {{ talking }}
- dest:string
marcel.grosjean@unine.ch

UPDATE

Figure 46: Rule action page

4.2.10 Graph explorer

The graph explorer in Figure 47 allows us to have a spatial and graphical view of all the elements present in the enclave in order to operate the rule engine.

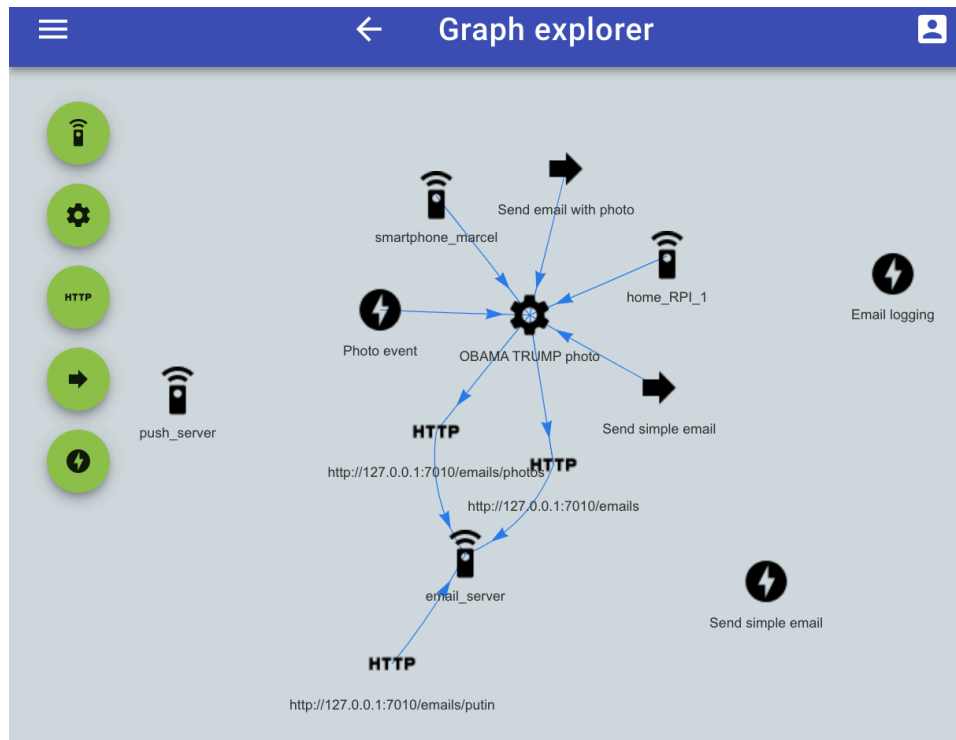


Figure 47: Graph explorer

As we can see, the *Smartphone_marcel* and *home_RPI_1* client can trigger the *OBAMA TRUMP photo* rule. This rule is linked to the *Photo event* event type and has two action types: *Send email with photo* and *Send simple email*. If this rule is triggered, then both the */emails* and */email/photo* URLs will be fired. These two URLs belong to the *email_server* client.

Thanks to this graphical explorer, we can be sure that our rule engine corresponds to the rules that have been decided beforehand.

4.3 Client scenario

4.3.1 Introduction

In this section, we will introduce the JavaScript client that was developed to test the enclave configured with our web application. This JavaScript client implements the same session creation and authentication protocol as a normal client or as RIOT. Its purpose is to capture information using two sensors. The first is the microphone that recognizes your spoken English. The second sensor is the camera that allows you to recognize a face and its facial expression.

In the previous part, we configured our enclave so that a rule is triggered if a person is not happy (*angry* > 0.5) and says in a sentence "*Obama*" or "*Trump*". The following scenario will be carried out with the client in order to test the correct configuration of the enclave by our RIOT interface:

- Open the client and configure the parameters as follows:
 - Client Id = 14, Event Type ID = 30, Email = marcel.grosjean@unine.ch, password = 123456
- Generate a public/private key keypair.
- Generate a shared secret.
- Login.
- Do not forget to turn on and authorize camera and microphone on the client.
- Stare at the camera while making an angry face and saying, "*I want to kill Donald Trump*".
- Check if the action was triggered.
- Check if the event was recorded.
- Check if the action message was recorded.

4.3.2 Client

The client in Figure 48 is a web client written in JavaScript. It is divided into three parts delimited by color.

The upper green part displays the content currently being filmed by the camera as well as a tracking device to identify in real time if a face is present. The *Client parameters* section also displays the basic parameters that will be used by our client, which are the ID client, the event type ID, the email and the password.

The green part of the upper middle with the title *Emotions...* corresponds to the emotions recognized by our classifier. There are the following 4 emotions: *Angry*, *Sad*, *Surprised* and *Happy*. We will only use the *angry* emotion. An emotion is quantified on a scale from 0 to 1.

The lower middle part displays the text spoken in English recognized by the microphone and the lower part performs a screenshot when text is identified.

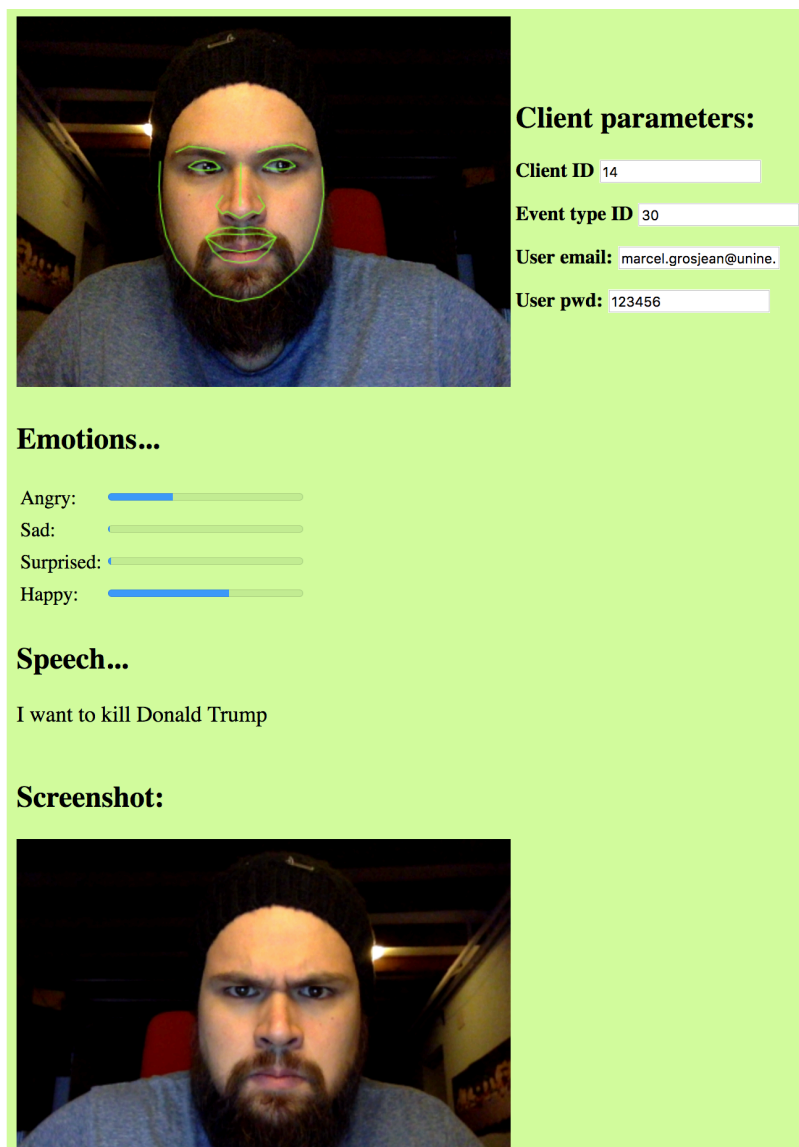


Figure 48: Client left part

The right part in Figure 49 is more technical and is divided into two parts. It is used to visually see which keys are used for encryption as well as to see which data are being sent and received by the client.

The yellow part allows you to perform three actions. Once the data in the green part has been correctly filled in, it is then possible to create a key pair with the *Gen ECDH key pair* button. Once this is done, it is possible to create a session with the enclave with the *Generate shared secret* button. Once completed, it is possible to connect with the *Login* button. Each step, if executed correctly, will display the result in the corresponding textarea.

The blue part shows which data are sent by the client (before encryption) and which data is received by the client and sent by the enclave.

Client public key:

MFkwEwYHkoZlZjOCAQYIKoZlZjO0DAQcDQgAEfyTisZHyCwa9eNckE6UsefxllcQigQkI4eMjrfL_GtiwUkyc3G0MYBuCMgTBZMDWPFHwmmQP05G_ypVNCj66Q

Enclave public key:

MFkwEwYHkoZlZjOCAQYIKoZlZjO0DAQcDQgAEBIWB-zQ8AqqlikiOhVleyBtxlZ7zozz8oLESoxrkaKD5knhRrJaYX7sVf2SiOZR8wM3BFVSitdcOxLNM-Njm-uA

Shared secret:

2PBvxKDn1f2v-wQicvZglCrOdFfPGvVrq0iSi1mLKM

JWT:

eyJraWQiOiJ0VEEJpWVdRM1pqWXRORFJoTXkwME9UWXdmVGhqWVdJdFlTTTBZekptT0RGaE1USm0iLCJhbGciOiJIUkEiLCJjaWQiOiJxNyJ9.LTMwMDEzNDc2OTM3MzM3MTU4MTU2MDkzMtG5ODY6.eyJtYWciOiI3ZGFiy2EzNi0xMzUwLTQzMWItOWI5My1kNzBiYmQ5ZTIiNjglLCJhZG1pbil6InRydWUil

Gen ECDH key pair

Generate shared secret

Login

Values sent to the enclave:

URL: POST:http://192.168.1.29:7001/enclave/1.0/events

BODY:

{
"clientid": 14,
"eventtypeid": 30,
"timestamp": "2018-11-01T11:23:00.861Z",
"properties": {
"talking": {
"type": "string",
"value": "I want to kill Donald Trump"
},
"angry": {
"type": "number",
"value": 0.603957140325086
},
"photo": {
"type": "string",
"value": "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAZAAAEsCAYAAADtt+XCAAAGAEIEQVR4Xuy925MI2XXetzLPyXNOVXX3X
}
}
}

Enclave response:

{ "data": { "insertedId": 461 }, "status": 201, "statusText": "", "headers": { "authorization": "eyJraWQiOiJ0VEEJpWVdRM1pqWXRORFJoTXkwME9UWXdmVGhqWVdJdFlTTTBZekptT0RGaE1USm0iLCJhbGciOiJIUkEiLCJjaWQiOiJxNyJ9.LTMwMDEzNDc2OTM3MzM3MTU4MTU2MDkzMtG5ODY6.eyJtYWciOiI3ZGFiy2EzNi0xMzUwLTQzMWItOWI5My1kNzBiYmQ5ZTIiNjglLCJhZG1pbil6InRydWUil" }

Figure 49: Client right part

An HTTP request is sent every time a sentence is identified. The following JSON shows an example of request.

```

778  {
779    "clientid": 14,
780    "eventtypeid": 30,
781    "timestamp": "2018-11-01T11:23:00.861Z",
782    "properties": {
783      "talking": {
784        "type": "string",
785        "value": "I want to kill Donald Trump"
786      },
787      "angry": {
788        "type": "number",
789        "value": 0.603957140325086
790      },
791      "photo": {
792        "type": "string",
793        "value": "data:image/png;base64,MahImageInBase64"
794      }
795    }
796  }

```

4.3.3 Emails sent

The actions to be taken are to send two emails. The first is a logging email while the second is a more complete email. As we can see in Figure 50, both emails were received in our inbox.

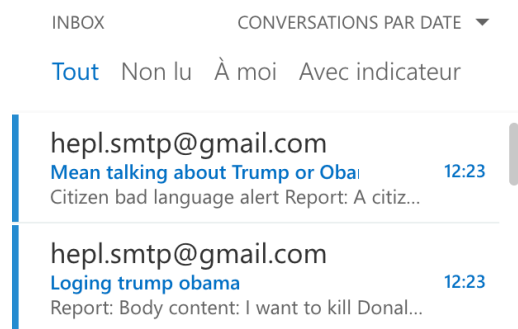


Figure 50: Mailbox after actions triggered

Figure 51 is the content of the logging email:

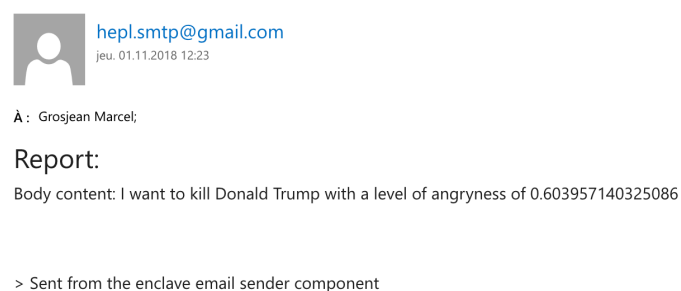


Figure 51: Logging email

Figure 52 is the content of the detailed email:

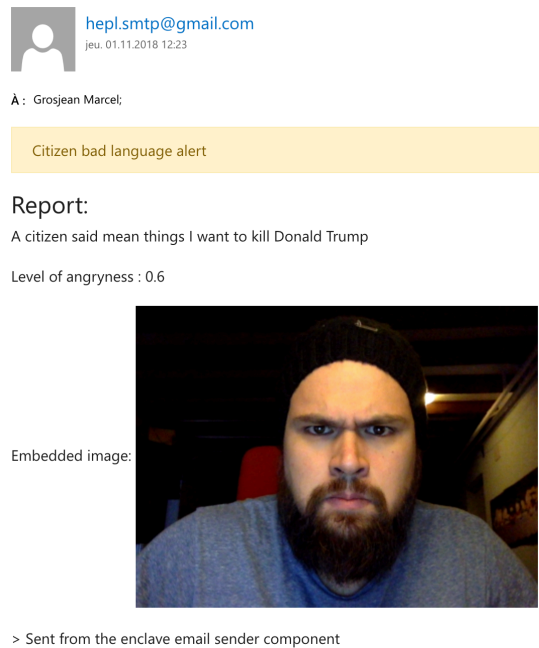


Figure 52: Detailed email

We find in this email all the elements present in the event type. The data transformations in the angryness level and in the text have been applied as shown in the illustration.

4.3.4 Events List

The list of events in Figure 53 allows you to list all the events that have occurred. Filters by date, clients and event type are applicable. It is also possible to fire an event directly without going through a client. The client can also directly delete an event. As we can see, our event sent by *smartphone_marcel* has been created.

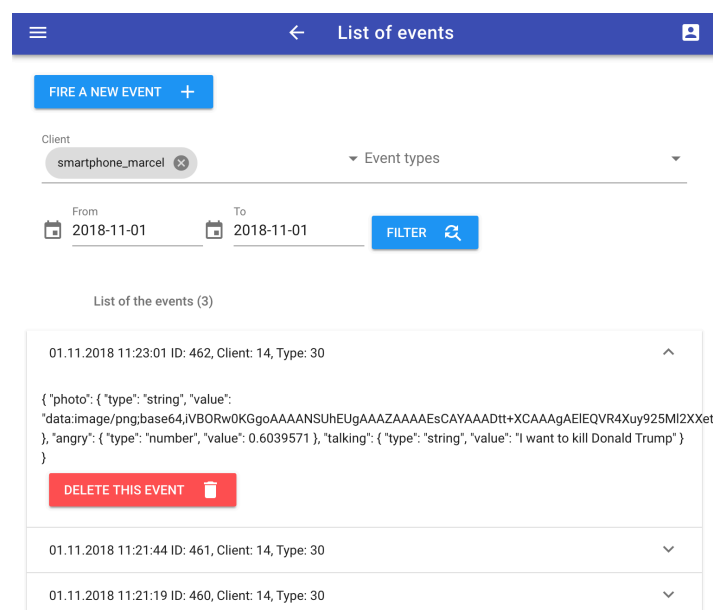


Figure 53: Events list page

4.3.5 Action message

Action messages in Figure 54 allow you to see which actions have been taken by a certain client. The content of the message is the same as the one sent to the action target. As we can see, the rule engine has performed the email sending actions correctly.

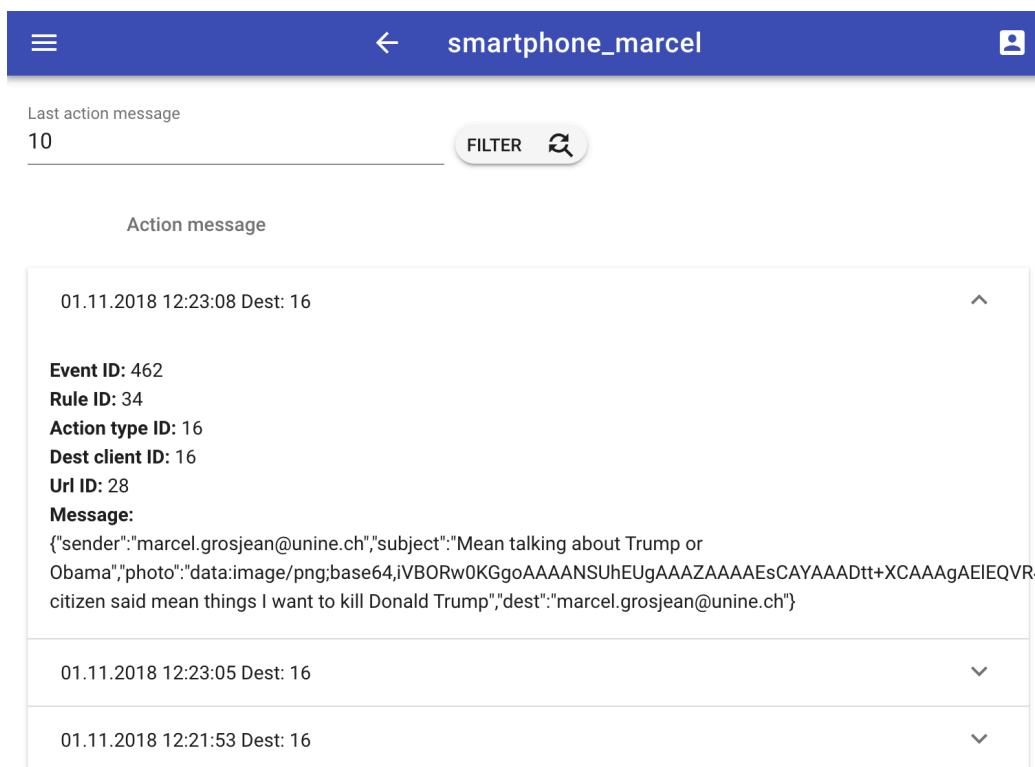


Figure 54: Action message page

4.4 Conclusion

This practical example shows that it is now possible to trigger actions on very complex and heterogeneous data. Indeed, we were able to trigger actions based on data from the camera and microphone. Thanks to a very weak coupling between Things and the enclave, and thanks to the notion of event type, it is then possible to connect a very large number of Things and perform actions according to the content sent. Practical limits are not limited to espionage, but can be applied to areas such as smart cities, health, finance, transport, etc. Practical and commercial applications are endless because nowadays the value is in the data and being able to exploit them in real time allows us to give real added value to data flows.

5

Administrative part

5.1 Introduction	122
5.2 Planification	122
5.3 Logbook	123
5.4 Burndown chart	123
5.5 Bibliography & Webography	128

5.1 Introduction

This master thesis is meant to show at an advanced and high level, the student's capacities and abilities for analysis, synthesis and critical thinking. Written, oral and communication skills must be shown in this work as well as scientific and technical approaches.

The first meeting with the project manager was in April 2018. It provided an opportunity to present the scope of the project and define the context and objectives to be achieved. The official start of the project is the first of June 2018. The official end of the project is the first of December 2018. It was planned to submit a report as well as the software artifacts developed during this period. During the project, no meetings are planned other than the usual email communications.

This work is worth 30 ECTS credits.

5.2 Planification

A GANTT diagram shows the overall task planning that takes place at the beginning of the project. Then, we mark the actual tasks in order to compare the original planning with the final planning.

Table 7: GANTT planification

Task	Type	June 2018				July 2018				August 2018				September 2018				October 2018				November 2018			
		1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Analyse ECDH + AES for Java	Planified																								
	Done																								
Implement secure protocol from enclave to backend	Planified																								
	Done																								
Analyse the whole architecture	Planified																								
	Done																								
Implement the frontend + enclave + email + Zuul	Planified																								
	Done																								
Test (front + backend + enclave)	Planified																								
	Done																								
Project documentation	Planified																								
	Done																								

5.3 Logbook

Table 8: Logbook

Date [dd.mm.yyyy]	Task	Time [hours]	Description
31.05.2018	Meeting	1	- Meeting with P.Gremaud for the project start
31.05.2018	Documentation	2	- Creation of the private git repos - Creation of the basic documentation structure for the project
01.06.2018	Research	3	- Looked for Java documentation on ECDH and AES
03.06.2018	Research	3	- Continuation of research documentation on ECDH and AES
03.06.2018	Development	4	- Project initialization for the enclave. Installation of -> spring boot -> MYSQL + JPA -> Bouncy Castle
03.06.2018	Documentation	1	Readme of the enclave
04.06.2018	Development	4	- Playing a bit with java.security => Trying to generate keys with ecdh -> Managed to generate public and private key -> Now trying to export public key as string and reimporting string key as PublicKey
06.06.2018	Development	6	- Still playing with ECDH. Managed to do some cool stuff -> Starting routes in spring boot -> some controllers -> Good exception handling -> Some Spring JDBC -> some testing with database pooling and hikaricp -> Some doc
07.06.2018	Development	6	Swagger 2 configuration + autogeneration for enclave JWT integration for enclave
08.06.2018	Development	1	Replacing the current jwt library by jose because the current lib does not support JWE
08.06.2018	Research	3	-> Research about JWE -> Research about Web Crypto API
09.06.2018	Research	2	Research about web crypto and how it could be used with standard crypto
09.08.2018	Research	6	Better organization of my java code for the cryptographic part Creation of a benchmark with java.crypto I managed to get rid of all third party cryptography package (no need of bouncy castle anymore) and used only java.security standard package
09.08.2018	Development	3	Developing the prototype for testing the web crypto API
09.08.2018	Documentation	1	Readme of the benchmark
15.08.2018	Development	10	Starting vuejs project and installing the following dependencies + configuration -> axios -> vuetify -> vue18n -> vue-router Configuration of the vueapp and login page
16.06.2018	Development	8	Got rid of the keys management on the enclave Adding the user management in the enclave Tester a bit mongodb Added the classes for the crypto to the enclave Added routes for authentication Started exchanging keys with the frontend Got rid of JPA Got rid of mongodb
17.06.2018	Research	2	Further research about JWE and analysing the protocol

17.06.2018	Development	4	Added redis for session management Started to play with frontend -> enclave connection
20.06.2018	Development	6	Better organization of my java code for the cryptographic part Creation of a benchmark with java.crypto I managed to get rid of all third party cryptography package (no need of bouncy castle anymore) and used only java.security standard package
21.06.2018	Documentation	1	-> Burndown chart -> Modify documentation of the project on github
21.06.2018	Development	4	-> limits the number of keys a client can create within one minute -> Add filter functionalities to turn object into standard response
23.06.2018	Research	5	Trying to understand and implement the new exchange protocol
25.06.2018	Research	6	Trying to implement the new protocol, failed, I decided to stick to a simpler version of the protocol
26.06.2018	Research	12	Trying to make web crypto work with Keys that comes from Java. Not obvious at all !!
27.06.2018	Development	3	I decided to give up the full key exchange protocol and stick with a simpler version I managed to import the java keys in webcrypto
27.06.2018	Development	5	I managed to derive the public key from java in web crypto. I had issues because: -> Web crypto doesn't like padding and java don't care => leads to many problems -> To derive a keys there are many steps in web crypto and it's not that obvious -> web crypto is not very verbose when there is an exception but i managed to derived a key and decrypt messages from the enclave
28.06.2018	Development	9	-> derived key in web crypto -> fix IV generation from enclave -> import correct IV and increment it in web crypto -> Encrypt data with sharedsecret in webcrypto -> post encrypted data from web crypto et enclave and decrypt data in enclave -> Refactor code
29.06.2018	Development	4	-> added routes to manage the keypair in the enclave -> added 2 new windows to the frontend -> list all valid keypair -> see the details of a keypair and delete it
30.06.2018	Development	5	CRUD for user accounts with frontend input
02.07.2018	Development	10	-> Found out that the library that I used does not allow the usage of hmac for signature so I changed to nimbus library for JWE/JWT/WS tokens generation -> Crypt JWE in JWT with nimbus and trying to decrypt in the frontend -> Continuing the filter and axios interceptor middleware
03.07.2018	Research	1	Some research about Spring Zuul
03.07.2018	Development	4	-> Fixed problems with CORS and returning custom headers -> modifying request filter for decryption (not finished yet) -> Axios interceptor does not like promises, had to modify and use await/async
04.07.2018	Development	5	Trying to make filter request wrapper works but it still does not work !!!!!!! Thinking about finding a new solution before sending ciphered data to the controller
05.07.2018	Development	3	Made attempts to make Filter request wrapper workd correctly but it doesn't
05.07.2018	Research	2	Made further research about Netflix Zuul
05.07.2018	Development	2	Developped a simple gateway with zuul and it seem to work !!!
06.07.2018	Development	7	-> Decrypt payload in gateway and replace with original payload and send it to the enclave -> Create custom java annotation for Authorization for each request -> The NIMBUS library for generating JWE doesn't really meet my requirements, I decided to develop my own JWE library

07.07.2018	Development	9	-> Finished to develop my own library to create and verify JWE -> I made authentication in enclave filter -> I made custom AUTHORIZATION with custom Java Annotation for each route based on user role
08.07.2018	Development	8	I made (or try to make working) the chiptography gateway with spring zuul, many errors
09.07.2018	Development	8	I managed to make the gateway with Zuul works and there were a big with the javascript library when increasing biginteger number. -> I found another library that works when increasing really huge numbers
11.07.2018	Development	1	Correction of some bugs
11.07.2018	Research	2	Read articles on iflux and wrote some comments
12.07.2018	Research	1	Read some articles about iflux and try to install iflux but not enough disk space xD
12.07.2018	Development	4	Change some parts of the base UI
13.07.2018	Development	3	Fixed some bugs in the UI while trying to use the kid stored in the localStorage
14.07.2018	Development	6	-> Continued to fix the kid problem in localStorage -> Added some ui to manage user own settings and better display of the informations according to user role
15.07.2018	Development	5	-> Fixed bug in frontend -> update some ui component due to the new version of vuetify -> Started CRUD for clients
16.07.2018	Development	4	-> Clients CRUD Done -> Fixed a bug with swagger and the servlets filter
17.07.2018	Development	4	-> Started events and eventtype
18.07.2018	Development	5	-> Finished CRUD events type
20.07.2018	Development	8	-> Remade CRUD events type and validation according to the comments of Pascal
21.07.2018	Development	6	-> Events CRUD with custom filter
22.07.2018	Development	4	-> Urls CRUD -> Some modifications in enclave code structure
23.07.2018	Development	2	-> Started CRUD Actions types
24.07.2018	Development	4	-> Finished CRUD Actions types
26.07.2018	Research	2	-> research about iflux
27.07.2018	Research	3	-> still researching about iflux
28.07.2018	Development	4	-> better filter for events, action type and event type
03.08.2018	Development	5	-> better ergonomoy -> added url and link from url to client in graph
03.08.2018	Documentation	3	-> github enclave doc -> made a global burndown chart instead of multiple one
04.08.2018	Development	6	-> Started the Rules add interface
05.08.2018	Development	8	-> Finished the client rules add interface -> Started rules controller
06.08.2018	Development	5	-> Continued CRUD Rules
07.08.2018	Development	10	-> Finished CRUD Rules in frontend and enclave -> Added actiontype, eventtype and rules to the graph
08.08.2018	Development	4	-> Started processing the rules for incoming events. => java sucks for javascript eval...
11.08..2018	Development	5	-> Continued the rules and try to eval js
12.08..2018	Development	5	-> Finished rules evaluation

13.08.2018	Development	5	-> Started the email sender component with nodejs. Added routes, swagger doc, logging, route validation, exception handling and email sending
17.08.2018	Development	5	-> Started to make a client in python but decided finally to do it in javascript
18.08.2018	Development	5	-> Started to do a client in JavaScript
19.08.2018	Development	5	-> Continued client + added key exchange + added emotion and speech recognition
20.08.2018	Development	5	-> Try to send data from client to enclave, fix some errors
21.08.2018	Development	7	-> Managed to send data to the enclave and evaluate the function. -> Fixed timeout issues with RestTemplate -> Manage to send picture to the emailsender -> Started handlebar for email sending formatting
22.08.2018	Development	7	-> fixed many bugs -> Started action message
23.08.2018	Development	6	-> Finished action message -> Finished formatting mail with handlebarjs -> Some minors changes in the UI
24.08.2018	Documentation	1	-> Deployment diagram
24.08.2018	Development	4	-> Added active switch to rules -> Started modifying the rules interface and enclave because it does not match the swagger api provided by Pascal
25.08.2018	Development	2	-> Added active status to rules (forgot to add it before) -> Continue multiple action to rule
26.08.2018	Development	6	-> Continued multiple action for a rule
28.08.2018	Development	6	-> Finished multiple action for a rule -> Changed the events rule engine in order to manage several rules for an event
29.08.2018	Documentation	2	-> Created a video to show how developed is the project
29.08.2018	Documentation	3	-> Started to see how i'm going to do the doc
10.09.2018	Documentation	3	Report
18.09.2018	Documentation	6	Report
19.09.2018	Documentation	6	Report
21.09.2018	Documentation	12	Report
22.09.2018	Documentation	12	Report
23.09.2018	Documentation	5	Report
24.09.2018	Documentation	5	Report
26.09.2018	Documentation	2	Report
28.09.2018	Documentation	3	Report
29.09.2018	Documentation	5	Report
30.09.2018	Documentation	5	Report
01.10.2018	Documentation	7	Report
02.10.2018	Documentation	6	Report
03.10.2018	Documentation	5	Report
04.10.2018	Documentation	3	Report
05.10.2018	Documentation	2	Report

5 Administrative part

06.10.2018	Documentation	5	Report
08.10.2018	Documentation	13	Report
10.10.2018	Documentation	5	Report
11.10.2018	Documentation	5	Report
13.10.2018	Documentation	5	Report
14.10.2018	Documentation	5	Report
15.10.2018	Documentation	5	Report
16.10.2018	Development	2	Errors logging and bug fixes
16.10.2018	Documentation	7	Report
17.10.2018	Documentation	9	Report
19.10.2018	Documentation	3	Report
20.10.2018	Documentation	3	Report
22.10.2018	Development	2	Some development and bud fixes
22.10.2018	Documentation	7	Report
23.10.2018	Documentation	10	Report + readme + starting dockerize infrastructure
24.10.2018	Documentation	5	Continued dockerizing everything
26.10.2018	Documentation	10	Continued again to dockerize everything. Fix bug with database and docker mysql. Docker brrrrrrr
27.10.2018	Documentation	8	Report
28.10.2018	Documentation	10	Report
30.10.2018	Documentation	6	Report
31.10.2018	Documentation	4	Report
01.11.2018	Documentation	10	Report
03.11.2018	Documentation	5	Report
04.11.2018	Documentation	1	Report
05.11.2018	Documentation	12	Report
06.11.2018	Documentation	1	Report correction with antidote 9
07.11.2018	Documentation	2	Mini ppt presentation for demo
08.11.2018	Meeting	1	Demo at unifr
10.11.2018	Documentation	1	Report
11.11.2018	Documentation	5	Report & ZIP containing the whole thesis

5.4 Burndown chart

The burndown chart is a graph that shows the progress of the actual working time of a project compared to the planned schedule. The project lasts a total of nine hundred hours and ends when these nine hundred hours have elapsed. The blue line shows the elapsed time from nine hundred to zero. The developer must follow this blue line by providing an adequate volume of hours. The red line shows the actual work done by the developer. If the red line is above, it means that the developer has not worked enough hours. If the red line is below the blue line, it means that the developer is ahead of schedule.

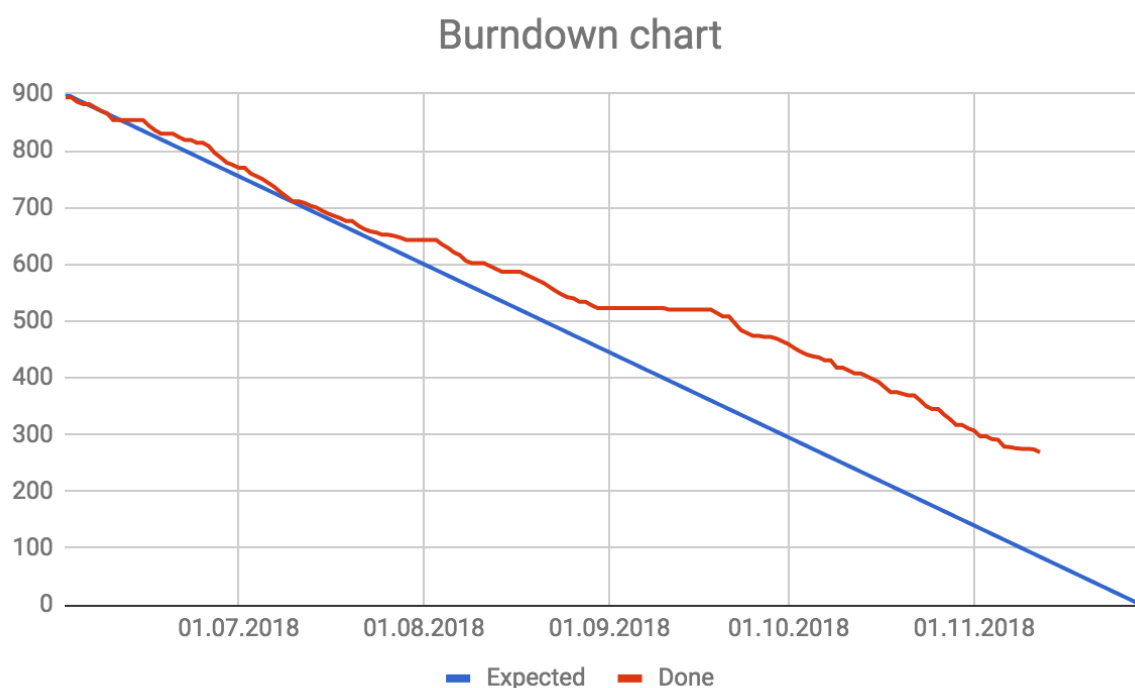


Figure 55: Burndown chart

As we can see, the actual work was constant at the beginning of the project. If we compare this chart to the GANTT, the first part represents the cryptography part. Then a short break was taken before starting the development of the infrastructure, which will last until the middle of the project. This pause will increase the delay in relation to the blue line. Then a second break was taken between the end of the development before starting the report. This second break will again increase the delay compared to the blue line. The red line is not finished because the project ended before the official planned end.

The actual working time for this master's work amounts to six hundred and eleven hours (611), or 70.11% of the total work requested.

5.5 Bibliography & Webography

- [1] P. Gremaud, A. Durand and J. Pasquier, "A secure, privacy-preserving IoT middleware using Intel SGX," *ACM*, no. Proceedings of the Seventh International Conference on the Internet of Things, 2017.
- [2] A. Hevner, S. March, J. Park and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly*, 2004.
- [3] Wikipedia, "Internet of Things," [Online]. Available: https://en.wikipedia.org/wiki/Internet_of_things. [Accessed 22 09 2018].
- [4] P. Gremaud, "Trustno1," [Online]. Available: <https://github.com/polchky/Trustno1-doc>. [Accessed 01 October 2018].
- [5] O. Liechti, L. Prévost, V. Delaye, J. Hennebert, V. Grivel, J.-P. Rey, J. Depraz and M. Sommer, "Enabling reactive cities with the iFLUX middleware," *WoT '15, Seoul, Republic of Korea*, no. ACM 978-1-4503-4045-8/15/10, 2015.
- [6] IFTTT, "IFTTT Platform documentation," [Online]. Available: <https://platform.ifttt.com/docs>. [Accessed 23 09 2018].
- [7] Intel, "Intel Software Guard Extensions (Intel SGX)," [Online]. Available: <https://software.intel.com/en-us/sgx>. [Accessed 29 September 2018].
- [8] Intel, "Introduction to Intel Software Guard Extensions webinar," 18 April 2017. [Online]. Available: <https://software.intel.com/sites/default/files/managed/81/61/intel-sgx-webinar.pdf>. [Accessed 29 September 2018].
- [9] Wikipedia, "Trusted execution environment," [Online]. Available: https://en.wikipedia.org/wiki/Trusted_execution_environment. [Accessed 28 09 2018].
- [10] R. Hayton, "Trusted execution environments: What, how and why?," *IOT Agenda*, 18 April 2018. [Online]. Available: <https://internetofthingsagenda.techtarget.com/blog/IoT->

- Agenda/Trusted-execution-environments-What-how-and-why. [Accessed 29 September 2018].
- [11] Wikipedia, "Diffie-Hellman key exchange," [Online]. Available: https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange. [Accessed 01 October 2018].
- [12] Wikipedia, "Advanced Encryption Standard," [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard. [Accessed 01 October 2018].
- [13] A. Ansel, "L'algorithme d'échange de clés Diffie-Hellman," 29 Fébruary 2016. [Online]. Available: <https://medium.com/@antoine.ansel/l-algorithme-d-%C3%A9change-de-cl%C3%A9s-diffie-hellman-6f9681d1418c>. [Accessed 01 October 2018].
- [14] A. Corbellini, "Elliptic Curve Cryptography: ECDH and ECDSA," 30 May 2015. [Online]. Available: <http://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/>. [Accessed 01 October 2018].
- [15] M. David and V. John, "The Galois/Counter Mode of Operation (GCM)," [Online]. Available: <http://luca-giuzzi.unibs.it/corsi/Support/papers-cryptography/gcm-spec.pdf>. [Accessed 02 October 2018].
- [16] N. Madden, "Ephemeral elliptic curve Diffie-Hellman key agreement in Java," 20 May 2016. [Online]. Available: <https://neilmadden.blog/2016/05/20/ephemeral-elliptic-curve-diffie-hellman-key-agreement-in-java/>. [Accessed 02 October 2018].
- [17] M. Rouse, "Advanced Encryption Standard (AES)," March 2017. [Online]. Available: <https://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard>. [Accessed 02 October 2018].
- [18] S. Info, "L'AES : Advanced Encryption Standard," 4 November 2001. [Online]. Available: <https://www.securiteinfo.com/cryptographie/aes.shtml>. [Accessed 02 October 2018].

- [19] N. Coffey, "Removing the 128-bit key restriction in Java," 2012. [Online]. Available: https://www.javamex.com/tutorials/cryptography/unrestricted_policy_files.shtml. [Accessed 09 October 2018].
- [20] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," NIST, U.S. Department of Commerce, November 2007.
- [21] P. Siriwardena, "JWT, JWS and JWE for Not So Dummies!," 26 April 2016. [Online]. Available: <https://medium.facilelogin.com/jwt-jws-and-jwe-for-not-so-dummies-b63310d201a3>. [Accessed 03 October 2018].
- [22] W3C, "Web Crypto API," 29 September 2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API. [Accessed 09 October 2018].
- [23] J. Tan, "Update on Web Cryptography," 21 July 2017. [Online]. Available: <https://webkit.org/blog/7790/update-on-web-cryptography/>. [Accessed 09 October 2018].
- [24] Encryb, "Comparing Performance of JavaScript Cryptography Libraries," 09 Jun 2015. [Online]. Available: <https://medium.com/@encryb/comparing-performance-of-javascript-cryptography-libraries-42fb138116f3>. [Accessed 09 October 2018].
- [25] Oracle, "Package java.security," [Online]. Available: <https://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html>. [Accessed 09 October 2018].
- [26] J. Kunter, "Using Netflix Zuul to Proxy your Microservices," 02 March 2016. [Online]. Available: https://blog.heroku.com/using_netflix_zuul_to_proxy_your_microservices. [Accessed 11 October 2018].
- [27] S. P. R. Katamreddy, "Why Spring Boot ?," 20 May 2016. [Online]. Available: <https://dzone.com/articles/why-springboot>. [Accessed 14 October 2018].
- [28] S. Khillar, "Difference between Authentication and Authorization," 30 October 2017. [Online]. Available: Difference between Authentication and Authorization. [Accessed 15 October 2018].

- [29] B. Leite, "Guide to Spring Boot REST API Error Handling," [Online]. Available: <https://www.toptal.com/java/spring-boot-rest-api-error-handling>. [Accessed 16 October 2018].
- [30] Xplenty, "The SQL vs NoSQL Difference: MySQL vs MongoDB," 28 September 2017. [Online]. Available: <https://medium.com/xplenty-blog/the-sql-vs-nosql-difference-mysql-vs-mongodb-32c9980e67b2>. [Accessed 16 October 2018].
- [31] T. Kadlecsek, "Node.js + MySQL Example: Handling 100's of GigaBytes of Data," 06 June 2017. [Online]. Available: <https://blog.risingstack.com/node-js-mysql-example-handling-hundred-gigabytes-of-data/>. [Accessed 16 October 2018].
- [32] T. D. C. S. M. P. J. B. Oliver Gierke, "Spring Data JPA - Reference Documentation," 15 October 2018. [Online]. Available: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>. [Accessed 16 October 2018].
- [33] E. Korotya, "5 Best JavaScript Frameworks in 2017," 17 January 2017. [Online]. Available: <https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282>. [Accessed 06 October 2018].
- [34] W. Woodhead, "Should you use Material Design?," 11 April 2018. [Online]. Available: <https://medium.com/pilcro/should-you-use-material-design-bfb596a04bae>. [Accessed 08 October 2018].
- [35] S. J. Vaughan-Nichols, "What is Docker and why is it so darn popular?," 21 March 2018. [Online]. Available: <https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/>. [Accessed 27 October 2018].
- [36] G. Selander, J. Mattsson and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)," 25 April 2017. [Online]. Available: <https://tools.ietf.org/html/draft-selander-ace-cose-ecdhe-06>. [Accessed 28 October 2018].
- [37] M. Fowler, Uml Distilled: A Brief Guide to the Standard Object Modeling Language, ??: Addison-Wesley, 2000.

6

Appendix

6.1 Cryptography benchmark Readme	134
6.2 Front-end Readme	140
6.3 Enclave Readme	142
6.4 Zuul Gateway Readme	147
6.5 Email sender Readme	151

6.1 Cryptography benchmark Readme

Benchmark: Java.security vs Web Crypto API

This repository compares the performances of both Java and Web Crypto API for generating ECDH keys, encode and decode messages with AES-GCM algorithms.

Java uses java.security which is the Java default security package. It does not implement anything since it's only a wrapper for third party implementation. The default implementation used is the one made by Sun Microsystems. A third party implementation like OpenSSL can be easily specified. We won't use any other third party library.

The Web Crypto API is an interface allowing javascript to use cryptographic primitives directly implemented in modern browsers. It's supposed to be much faster than pure JavaScript security library.

We're going to compare both java.security and Web Crypto Api in terms of pure performance to see which is the fastest.

Requirements

- Java JDK 1.8
- A recent version of (Safari, Chrome, Firefox, Edge)

Installation

For the web crypto API you don't need to install anything apart from having the latest version of your browser.

We are using Java default library (java.security) that wraps Sun Microsystems primitives for encryption.

By default the maximum Java key size is 128 bits... Why ? Because some countries limit the key size at 128 bits and the java default library complies with theses limitations...

In order to override this limitation, you need to download the [extended Java Cryptography Policy](#) from Oracle. Then unzip the content and replace the content of the following directory with the unzipped content:

```
${java.home}/jre/lib/security/
```

Use case

In order to compare the performances, we need to do similar operations on both platform even though they are not similar at all.

Java usecases

Java is rather simple to monitor since the code is synchronous by design. We only need to measure the time elapsed from the beginning to the end.

1. Create ECDH instance for both Alice and Bob
2. Print to the console their keypair
3. Create AES instance for both Alice and Bob
4. Exchange public keys
5. Creating a shared secret and derive a key from it
6. Generate the iv
7. Start the process of exchanging messages. This process will be executed N times:
 - i. Defining a string message
 - ii. Alice crypts the message with her derived key
 - iii. The message is parsed to base64
 - iv. The message is parsed to string
 - v. The message is decrypted by Bob
 - vi. Increment the IV

The time elapsed is measured in seconds from before step 1 to after the last step of 7.

Javascript use cases

As Javascript is asynchronous and the Web Crypto API massively rely on Promises, it's therefore much harder to measure performances as things don't go in a deterministic order.

The following use case describe all the steps with the use of Promise and await/async.

1. Create the ECDH instance for Alice with Promise
2. Extract public key (private key is not extractable)
3. Create the ECDH instance for Bob with Promise
4. Extract public key (private key is not extractable)
5. Derive a key for Alice with Promise
6. Derive a key for Bob with Bob with Promise
7. Start the process of exchanging messages. This process will be executed N times: Since this process is asynchronous, we'll rely on await/async mechanism to pretend we're in a synchronous mode
 - i. Defining a string messages
 - ii. Generate the iv (new for each exchange)
 - iii. Crypt message with Alice derived key

iv. Decrypt message with Bob derived key

As you can see here, for the sake of simplicity we didn't increment the iv and we skipped the parse to Base64 process.

Results

We're going to see the results here.

The tests were executed on a Intel core i5 2.7Ghz Machine with 4Gb of RAM.

They both use:

- ECDH with 256 bits key
- AES-CGM 256
- A 12 bytes IV

Java security

In order to run the java application we need to install JDK 1.8 and do the importing step of the JCE. Otherwise we won't be able to run the app.

Before compiling the application we need to check the class **ch.tm.bench.App.java** and make sure that the following values are correct:

```
final boolean VERBOSE = false;
final int ITERATIONS = 1000000;
```

The simplest way to compile and run (due to classpath) is to use eclipse and run the application from there.

The result of the execution is:

```
Start benchmarking. Please wait...
```

```
Alice Private Key HEX      :
3041020100301306072A8648CE3D020106082A8648CE3D03010704273025020101042044B3F3826F19
A948080F1B0F5E59B89A7D38CFBC6B9A7696F21E4898F4F26CCB
Alice Private Key Base64 :
MEECAQAwEwYHKoZIzj0CAQYIKoZIzj0DAQcEJzAlAgEBBCBEs/OCbxmpSAgPGw9eWbiafTjPvGuadpbyHk
iY9PJsyw==
Alice public Key HEX      :
3059301306072A8648CE3D020106082A8648CE3D03010703420004809CB1C845DF75A504C6B1AC03F3
3D139DA99EEEA3E140433983C4D61CCF1D42CF25A7F296816ABB7B49AA644FAAF3E5FC19A632C66EA7
64CA3E18B71FAEBE68
Alice public Key Base64 :
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEgJyxyEXfdaUExrGsA/M9E52pnu6j4UBDOYPE1hzPHULPJa
fyloFqu3tJqmRPqvPl/BmmMsZup2TKPhi3H66+aA==
Alice shared secret HEX :
25265F7A8CD55EECD1DFD7BDDA26662F4F9702E512A989ACAA75C7E69B9544BB
```

```

Bob Private Key HEX      :
3041020100301306072A8648CE3D020106082A8648CE3D030107042730250201010420320D1236B245
9D75FF3093DAA9675595B42FBE18AB7365C965179DF7BB994D56
Bob Private Key Base64 :
MEECAQAwEwYHKoZIzj0CAQYIKoZIzj0DAQcEJzA1AgEBBCAyDRI2skWddf8wk9qpZ1WVtC++GKtzZc1lF5
33u5lNVg==
Bob public Key HEX      :
3059301306072A8648CE3D020106082A8648CE3D03010703420004E560F4DD90249D8A3CFE2C545791
D1344D8870D486B481E270D2CBC7C420761EF7CC3F881602B8C5E941CFD9E5C8B4C6AA238B852A2D66
2539100B2E112E16E5
Bob public Key Base64 :
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE5WD03ZAKnYo8/ixUV5HRNE2IcNSGtIHicNLLx8Qgdh73zD
+IFgK4xelBz9nlyLTGqiOLhSotZiU5EAsuES4W5Q==
Bob shared secret HEX :
25265F7A8CD55EECD1DFD7BDDA26662F4F9702E512A989ACAA75C7E69B9544BB

Mode verbose deactivated

Benchmark Finished:

The whole process took: 11.394719722 seconds
There were 1000000 iterations
Each iteration took in average 1.3394719722E-5 seconds

```

Web Crypto API

All you need to run the Web Crypto API demo is to open the bench.html file directly in the browser.

The tests were made with Chrome browser.

Once the page is opened, you need to click the button "Start computation" and wait for its completion. You must not click twice on this button otherwise things could go wrong...

Before starting the application, we need to make sure that the following variables are like below:

```

const VERBOSE = false;
const ITERATIONS = 1000000;

```

The result of the execution is:

```

Start Benchmarking. Please wait... 1000000 iterations have to run...

```

```

Alice key pair generation done. Not extractable.

```

```

Alice's public key: {"crv":"P-
256","ext":true,"key_ops":[],"kty":"EC","x":"SylqEc9eSva0jWYxh56P7Ko4EYFNf0jkP3_e0
ZLahgc","y":"s6sd_uDLrz2p0hyD1-1AgEbM2tuhc8GDfv71FQhjIOs"}

```

```

Bob key pair generation done. Not extractable.

```

```
Bob's public key: {"crv":"P-256","ext":true,"key_ops":[],"kty":"EC","x":"GkUMYeNs6WyTtpW9ySFuTBkPtHu3eyNPXpHiUvANPfs","y":"YU9hhXfrqtopKHfwHXvFWln0XwzhORHIL7gnSjv67Q0"}
```

Alice derived shared secret generation done. Not extractable.

Bob derived shared secret generation done. Not extractable.

Verbose mode deactivated

Benchmark Finished:

The whole process took: 239.148 seconds

There were: 1000000 iterations

Each iteration took in average 0.000239148 seconds

Conclusion

We can obviously see which one is the fastest here.

Java took 11 seconds to complete the whole process while JavaScript took 240 seconds. As a reminder, the process involves crypt/decrypt 1'000'000 messages with AES-CGM. Javascript has also less work to do as we skipped conversion from binary to Base64.

There is a 21-time speed up factor in favor of Java in this case!

Without further research I can easily guess that Promises and the use of async/await add overhead and massively slow down the process.

It would be good to see if there were a faster way to the process in Javascript.

Here are the pros and cons of both:

Java.security pros:

1. Lightning fast
2. Java.security wrapper is rather simple to use
3. Still portable and have many framework for REST (Spring,...)

Java.security cons:

1. Rely on third party implementation (can be obscure sometime).
2. JavaDoc still super difficult to use

Web Crypto pros:

1. Dead simple to use for simple usage

2. Super super portable since all browsers implement it (who don't update their browsers in 2018?)

Web Crypto cons:

1. Difficult to use for complex use case
2. No native support for BigInteger
3. Can't customize everything since most of the API is hidden to the developer
4. Slow due to the use of Promises

To sum up, I would still go with Java as it's lightning fast and beats Web crypto hands up. Performance is crucial because our project deals with Internet of things that by nature require high velocity.

Web Crypto API is still a good choice as long as we have a low velocity application.

6.2 Front-end Readme

Frontend for enclave

Project name: **RIOT** (Reactive Internet Of Things)

Requirements

In order to successfully install, run and build the application, you need to install the following dependency:

- NodeJS 8.5+

Installation of the app

You need to navigate to the root folder and type:

```
npm install
```

Technologies used

The following technologies and dependencies were used to develop the web client:

- VueJS 2
- Vuetify 1
- vue-router 3
- axios 0.18
- vue-i18n

Configuration of the application

All the constants are kept in one single file and need to be updated according the the production specs.

The constants file is located at:

```
./src/assets/js/consts.js
```

Run the development server

The nodemon service is called and is listening on all changes on the classpath. It will rebuild and reload the application after each update of the code.

```
npm run dev
```

Building the application for production

It will build the application and all the dependencies in order to run the application for production.

The app still needs to be hosted on a webserver like nginx.

```
npm run build
```

The built application is located at:

```
./dist
```

Running the application with Docker

It is also possible to run the application with Docker containers.

First we need to make sure that the **./Dockerfile** has all the correct values:

```
FROM node:9.11.1-alpine
RUN npm install -g http-server
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
CMD [ "http-server", "dist" ]
```

Then we need to build a docker image with the following command:

```
docker build -t frontend .
```

Then in the console type the following command to run the frontend in a docker image:

```
docker run -it -p 8080:8080 --rm --name dockerized-frontend frontend
```

The application should be available at: **http://localhost:8080**

6.3 Enclave Readme

TM-enclave

This repository contains the code for the enclave server. It will not be used as the real enclave but rather as a simulator that is used to develop the web frontend and compare the performances between Java and C++ for encryption.

Spring boot is used as the Java Framework. By default maven package manager is used to manage the packages. MySQL is used as the database.

REQUIREMENTS

- Java JDK 1.8
- Maven
- MySQL

Installation

This section describes all the steps to make the application running.

Extend default key size

We are using Java default library (java.crypto) that wraps third party libraries for encryption.

By default the maximum key size is 128 bits... Why ? Because some country limits the key size at 128 its and the java default library complies with theses limitations...

In order to override this limitation, you need to download the [extended Java Cryptography Policy](#) from Oracle. Then unzip the content and replace the content of the following directory with the unzipped content:

```
${java.home}/jre/lib/security/
```

Configuration

First you need to create the schema and import all the data in the database. All you need to do is to create the schema **enclave** in mysql and type the following command to import all the tables and data:

```
mysql -u root -p enclave < db.sql
```

You need to edit the file **src/main/resources/application.properties** and make sure that the following parameters are correct and match your local configuration in order to make run everything:

```
# Spring boot configuration
server.port=7000

# Application name
spring.application.name=enclave

# Cache request (necessary for not caching encryption)
spring.cache.type=NONE

# MySQL
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/enclave?useSSL=false
spring.datasource.username=root
#spring.datasource.password = "abc" # Keep commented if null

# Redis
spring.redis.host=localhost
spring.redis.port=6379
#spring.redis.password= ""
spring.redis.ssl=false
spring.cache.redis.cache-null-values=false

# Many many many stuff printed on the console
#logging.level.org.springframework = debug

# Hikaricp (jdbc pooling) conf
spring.datasource.hikari.connection-timeout=60000
spring.datasource.hikari.maximum-pool-size=5

# disable white page error
#server.error.whitelabel.enabled=false

# JWT
# Token expires after one hour
jwt.expireTimeSeconds=3600
jwt.secret=MahPrivateKeeey
# Defines the maximum ecdh key exchange we can do per minute with a certain IP
jwt.maximumCertificatesPerIp=1000

#Logging
logging.level.org.springframework.security= DEBUG
logging.file = logfile.log
```

Dependencies

In order to make everything work you first need to install all the dependencies. They're all managed by the maven. Go to the root directory and type:

```
mvn clean install
```

Compilation and execution

This section will describe how to compile and deploy the application.

Compile with maven

We'll use maven to compile the project. The jar will be generated by default in the **target/** directory.

If you want to configure the name of the jar file you need to go to the **pom.xml** file and configure the XML tags **name** and **version**.

By default the fat jar name is: **target/enclave-0.1.jar**

To compile and run the tests at the same time:

```
mvn package
```

Compile and run for development

Manual reloading takes time during development and executing the application with **java -jar** will run in production mode. This mode will ignore all changes in classpath.

In order to allow the autoreload we need to make sure that the following dependency is found in the **pom.xml** file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <scope>runtime</scope>
</dependency>
```

Now to trigger autoreload when code changes you need to have a look at your IDE documentation. (see eclipse, netbeans or intellij documentation)

Or if you want to reload manually you can simply type the command:

```
mvn package -T 1C -DskipTests && java -jar target/enclave-0.1.jar
```

The link to the Swagger documentation is:

```
http://localhost:7000/swagger-ui.html#/
```

Run as a service with systemd

We can run the enclave as a long-running service with systemd.

First we need to create a file with the following command:

```
touch /lib/systemd/system/enclave.service
```

This file contains the following informations:

```
[Unit]
Description=Enclave demo service
After=network.target # Enclave is executed after the network is ready
StartLimitIntervalSec=0
[Service]
Type=simple
Restart=always
RestartSec=1
User=root # user that executes the Enclave
# Command to run the application
ExecStart=/usr/bin/env java -jar /path/to/enclave-0.1.jar
```

```
[Install]
WantedBy=multi-user.target
```

And the following command is used to enable the service at startup and run the enclave:

```
systemctl enable enclave
systemctl start enclave
```

Running the application with Docker

It is also possible to run the application with Docker containers.

First of all, you need to update the JCE files by following the instructions are above.

We're going to run Redis and MySQL in their own Docker container, but Spring can't connect directly to the container using the loopback address. We need to modify the **application.properties** file and update the Redis and MySQL conf like the following:

```
# MySQL
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://mysql:3306/enclave?allowPublicKeyRetrieval=true
&useSSL=false
spring.datasource.username=dbuser
spring.datasource.password=dbpass

# Redis
spring.redis.host=redis
spring.redis.port=6379
#spring.redis.password= ""
spring.redis.ssl=false
spring.cache.redis.cache-null-values=false
```

Then we need to recompile our enclave:

```
mvn package -T 1C -DskipTests
```

We need to make sure that a Redis container (not persistent) is up and running:

```
docker run -it -p 6379:6379 --rm --name dockerized-redis redis
```

Then we need to make sure that a MySQL container (not persistent) is up and running:

```
docker run -it --rm -p 3306:3306 --name=dockerized-mysql -e
MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=enclave -e MYSQL_USER=dbuser -e
MYSQL_PASSWORD=dbpass mysql
```

It's time now to import the database (from the console in the root directory):

```
docker exec -i dockerized-mysql mysql -uroot -proot enclave < db/db.sql
```

We need to make sure that the **/Dockerfile** has all the correct values:

```
FROM java:8
VOLUME /tmp/enclave
EXPOSE 7000
ADD /target/enclave-0.1.jar enclave-0.1.jar
ENTRYPOINT ["java","-jar","enclave-0.1.jar"]
```

Then we need to build a docker image with the following command:

```
docker build -t enclave .
```

Then in the console type the following command to run the enclave in a docker container with a link to the Redis, MySQL and emailsender containers:

```
docker run -it --rm --link dockerized-redis:redis --link dockerized-mysql:mysql --
link dockerized-emailsender:emailsender -p 7000:7000 --name dockerized-enclave
enclave
```

If the action target is located in the emailsender container, we need to specify the following url in the RIOT interface:

```
#example of url using a docker container with the RIOT web interface.
http://emailsender:7010/emails
```

If we don't use docker container, we can replace emailsender by the ip or hostname of the action target.

The API should be available at: **http://localhost:7000**

6.4 Zuul Gateway Readme

TM-gateway

This repository contains the code for the gateway cryptography middleware. This middleware is used as a reverse proxy to catch HTTP request and use common cryptography technique in order to uncipher incoming requests and cipher outgoing requests.

The Spring Zuul implementation of Netflix Zuul is used as the reverse proxy. By default maven package manager is used to manage the packages. Redis is used as the database.

Requirements

- Java JDK 1.8
- Redis
- Maven

Installation

This section describes all the steps to make the application running.

Extend default key size

We are using Java default library (java.crypto) that wraps third party libraries for encryption.

By default the maximum key size is 128 bits... Why ? Because some country limits the key size at 128 its and the java default library complies with theses limitations...

In order to override this limitation, you need to download the [extended Java Cryptography Policy](#) from Oracle. Then unzip the content and replace the content of the following directory with the unzipped content:

```
${java.home}/jre/lib/security/
```

Configuration

First you need to edit the file **src/main/resources/application.properties** and make sure that the following parameters are correct and match your local configuration in order to make run everything:

```
# Url of the enclave
zuul.routes.enclave.url=http://localhost:7000

ribbon.eureka.enabled=false
```

```
# Port of the reverse proxy
server.port=7001

# Redis
spring.redis.host= localhost
spring.redis.port= 6379
#spring.redis.password= ""
spring.redis.ssl= false
spring.cache.redis.cache-null-values=false

#jwt
jwt.secret=MahPrivateKeeey
```

Secondly you need to edit the file **src/main/resources/application.yml** and make sure that the following parameters are correct and match your local configuration in order to allow cross domain queries and disable caching:

```
zuul.ignored-headers: Access-Control-Allow-Credentials, Access-Control-Allow-Origin

zuul.ignoreSecurityHeaders: false

zuul:
  sensitive-headers: Cookie, Set-Cookie
```

Dependencies

In order to make everything work you first need to install all the dependencies. They're all managed by the maven. Go to the root directory and type:

```
mvn clean install
```

Compilation and execution

This section will describe how to compile and deploy the application.

Compile with maven

We'll use maven to compile the project. The jar will be generated by default in the **target/** directory. If you want to configure the name of the jar file you need to go to the **pom.xml** file and configure the XML tags **name** and **version**.

By default the fat jar name is: **target/gateway-0.1.jar**

To compile and run the tests at the same time:

```
mvn package
```

Compile and run for development

Run the following command in order to compile and run the application.

```
mvn package -T 1C -DskipTests && java -jar target/gateway-0.1.jar
```

Run as a service with systemd

We can run the gateway as a long-running service with systemd.

First we need to create a file with the following command:

```
touch /lib/systemd/system/gateway.service
```

This file contains the following informations:

```
[Unit]
Description=Gateway demo service
After=network.target # Gateway is executed after the network is ready
StartLimitIntervalSec=0
[Service]
Type=simple
Restart=always
RestartSec=1
User=root # user that executes the gateway
# Command to run the application
ExecStart=/usr/bin/env java -jar /path/to/gateway-0.1.jar

[Install]
WantedBy=multi-user.target
```

And the following command is used to enable the service at startup and run the enclave:

```
systemctl enable gateway
systemctl start gateway
```

Running the application with Docker

It is also possible to run the application with Docker containers.

First of all, you need to update the JCE files by following the instructions are above.

We're going to run Redis in its own Docker container, but Spring can't connect directly to the container using the loopback address. We need to modify the **application.properties** file, update the enclave URL and the Redis conf like the following:

```
#Url Enclave
zuul.routes.enclave.url=http://enclave:7000

# Redis
spring.redis.host=redis
spring.redis.port=6379
#spring.redis.password= ""
spring.redis.ssl=false
spring.cache.redis.cache-null-values=false
```

Then we need to recompile our gateway:

```
mvn package -T 1C -DskipTests
```

We need to make sure that a Redis container (not persistent) is up and running:

```
docker run -it -p 6379:6379 --rm --name dockerized-redis redis
```

We need to make sure that the **/Dockerfile** has all the correct values:

```
FROM java:8
VOLUME /tmp/gateway
EXPOSE 7001
ADD /target/gateway-0.1.jar gateway-0.1.jar
ENTRYPOINT ["java", "-jar", "gateway-0.1.jar"]
```

Then we need to build a docker image with the following command:

```
docker build -t gateway .
```

Then in the console type the following command to run the gateway in a docker container with a link to the Redis and enclave containers:

```
docker run -it --rm --link dockerized-redis:redis --link dockerized-
enclave:enclave -p 7001:7001 --name dockerized-gateway gateway
```

The gateway should be available at: **http://localhost:7001**

6.5 Email sender Readme

Email sender

This server is an action target that is meant to act as an email sender. It exposes a REST API that is known and callable by the enclave.

Requirements

In order to successfully install, run and build the application, you need to install the following dependency:

- NodeJS 8.5+

Installation of the app

You need to navigate to the root folder and type:

```
npm install
```

Technologies used

The following technologies and dependencies were used to develop the email sender component:

- VueJS 2
- Express 4
- Swagger 3
- Winston 3
- Nodemailer 4
- HandlebarJS 3

Configuration of the application

There is a single configuratio file that contains all the routes and configuration properties. Before running the server, it is necessary to configure on which port it will run in the following file:

```
./index.js
```

Run the development server

The nodemon service is called and is listening on all changes on the classpath. It will rebuild and reload the application after each update of the code.

```
npm run dev
```

Building the application for production

We can run the email sender as a long-running service with systemd.

First we need to create a file with the following command:

```
touch /lib/systemd/system/emailsender.service
```

This file contains the following informations:

```
[Unit]
Description=Email Sender demo service
After=network.target # Emailsender is executed after the network is ready
StartLimitIntervalSec=0
[Service]
Type=simple
Restart=always
RestartSec=1
User=root # user that executes the Email Sender
# Command to run the application
ExecStart=/usr/bin/env node /path/to/index.js
```

```
[Install]
WantedBy=multi-user.target
```

And the following command is used to enable the service at startup and run the enclave:

```
systemctl enable emailsender
systemctl start emailsender
```

Running the application with Docker

It is also possible to run the application with Docker containers.

First we need to make sure that the **./Dockerfile** has all the correct values:

```
FROM node:8

RUN npm install -g nodemon

WORKDIR /frontend

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 7010

CMD [ "npm", "start" ]
```

Then we need to build a docker image with the following command:

```
docker build -t emailsender .
```

Then in the console type the following command to run the frontend in a docker image:

```
docker run -it -p 7010:7010 --rm --name dockerized-emailsender emailsender
```

The application should be available at: **<http://localhost:7010>**

.