Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Bartosz Tracz
Eirik Persson Masteholtet
Anders Isaksen

An AWS IoT-based system for handling communication and controlling an AGV

Bachelor's project in Bachelor of Engineering in Computer Science
Supervisor: Hao Wang
May 2019

NTNU
Kunnskap for en bedre verden

Bartosz Tracz
Eirik Persson Masteholtet
Anders Isaksen

An AWS IoT-based system for handling communication and controlling an AGV

Bachelor's project in Bachelor of Engineering in Computer Science
Supervisor: Hao Wang
May 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

An AWS IoT-based system for handling communication and controlling an AGV

Author(s)
Bartosz Tracz
Eirik Persson Masteholtet
Anders Isaksen

Bachelor of Science in Engineering - Computer Science
20 ECTS
Department of Computer Science
Norwegian University of Science and Technology,

20.05.2019

Supervisor Hao Wang

Sammendrag av Bacheloroppgaven

Tittel:	**AWS IoT-basert system for håndtering av kommunikasjon og kontroll av AGV**
Dato:	20.05.2019
Deltakere:	Bartosz Tracz Eirik Persson Masteholtet Anders Isaksen
Veiledere:	Hao Wang
Oppdragsgiver:	Escio AS, Terje Krogstad
Kontaktperson:	Terje Krogstad, terje.krogstad@escio.no
Nøkkelord:	Bachelor, API, IoT, AWS, REST, MQTT, Z-Wave, AGV
Antall sider:	59
Antall vedlegg:	6
Tilgjengelighet:	Åpen

Sammendrag:

Internet of Things (IoT) har de siste årene erfart en kraftig vekst. Denne veksten har ført til økt popularitet og tilgjengelighet av IoT-baserte løsninger på markedet. Robotiske teknologier, slik som Automated guided vehicles (AGV), er den neste bølgen av IoT-enheter som kan brukes til ulike industrielle formål. I lys av disse teknologiske fremskrittene, samt den foreslåtte oppgavebeskrivelsen utlevert av Escio AS, har vi implementert en prototype bestående av to deler. Den første delen omhandler utviklingen av et Representational State Transfer (REST) application programming interface (API) som sender data direkte til Amazon Web Services (AWS). I den andre delen ble det implementert et API basert på Z-Wave protokollen som sender data via MQ Telemetry Transport (MQTT) protokollen til AWS som tillater kommunikasjon med en AGV-enhet. Konklusjonen er at det er mulig og gjennomførbart å kontrollere en AGV-enhet basert på sensordata fra et IoT-nettverk, og at en slik løsning har stort potensial i virkelige scenarioer. Escio var veldig fornøyd med sluttproduktet.

Summary of Graduate Project

Title:	**An AWS IoT-based system for handling communication and controlling an AGV**
Date:	20.05.2019
Authors:	Bartosz Tracz Eirik Persson Masteholtet Anders Isaksen
Supervisor:	Hao Wang
Employer:	Escio AS, Terje Krogstad
Contact Person:	Terje Krogstad, terje.krogstad@escio.no
Keywords:	Thesis, Bachelor, API, IoT, AWS, REST, MQTT, Z-Wave, AGV
Pages:	59
Attachments:	6
Availability:	Open

Abstract:

In the recent years the growth of Internet of things (IoT) devices has increased exponentially. This growth has led to an increase in popularity and availability of devices on the market. Following the increase in popularity, the general use and development of IoT based solution has been in high demand. Robotic technologies such as automated guided vehicle (AGV), are the next wave of IoT devices that are used for various industrial purposes. Inspired by the advancements within these technologies and the need for a new solution requested by Escio AS, we have implemented a prototype providing a solution consisting of two parts. In the first part, we implemented a Representational State Transfer (REST) application programming interface (API) that sends data directly to the Amazon Web Services (AWS). In the second part, we developed an API based on the Z-Wave protocol that sends data through the MQ Telemetry Transport (MQTT) protocol to the AWS which controls an AGV unit. The conclusion is that it is possible and feasible to control an AGV unit based on sensor data from an IoT network, and that such a solution has great potential in real life scenarios. Escio was very pleased with the end product.

Preface

We would like to thank Escio AS for the opportunity to undertake this bachelor project. A special thanks to Hao Wang for supervising us throughout the development process and thesis phase. We would also like to thank Terje Krogstad from Escio AS for supervising us during the process and providing a great project experience.

Contents

List of Figures

List of Tables

Listings

Abbreviations

Abbreviation	Phrase
AGV	Automated Guided Vehicle
API	Application Programming Interface
ARM	Advanced RISC Machine
AWS	Amazon Web Services
CA	Certificate Authority
CLI	Command Line Interface
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
MAC	Media Access Control
MiR	Mobile Industrial Robots
MQTT	MQ Telemetry Transport
M2M	Machine to Machine
NDA	Non-Disclosure Agreement
ORM	Object-relational mapping
PDF	Portable Document Format
PEP	Python Enhancement Proposal
QOS	Quality of Service
REST	Representational State Transfer
SDK	Software Development Kit
SQL	Structured Query Language
SSH	Secure Shell
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
URL	Uniform Resource Locator
WSGI	Web Server Gateway Interface
XML	eXtensible Markup Language

1 Introduction

Internet of Things (IoT) is quickly asserting itself as one of the more dominant technologies on the marked. With an estimated number of devices between 6.4 and 17.6 billion, IoT strives to "connect the unconnected" using different paradigms and protocols. The technology is used in everything from home automation to deployment on a metropolitan scale and has received varying amounts of attention from a wide range of companies.

1.1 Motivation

As demand for various IoT products has increased, more companies want to try their hand at developing their own IoT-based solutions. Amongst these companies was Escio AS, who took an interest in this technology, and together with NT6 they have acquired an automated guided vehicle (AGV), specifically the MiR100. This device is designed to transport materials and logistics internally within production facilities, hospitals, warehouses, and a host of other industrial locations [5]. Escio and NT6 are currently looking at new scenarios for this technology, and is at the moment working on an application for 3D scanning buildings and automatically collecting data by driving a preconfigured route. Currently, the robot can be controlled via its web-based interface manually, but Escio seeks to develop a fully automated solution of the control process using Amazon Web Services (AWS) and real-time sensor data.

After doing some research, we found one related project using similar equipment and related ideas. The project, done by a group at Aalborg University in 2016, was to bring flexibility and automation to a part feeder in an Industry 4.0 context. This was an integrated solution with two standalone robots where the MiR100 was used as an integrated part. In the project they experimented with controlling the AGV with arrow keys by publishing to a certain topic and increasing the velocity based on key input. They also experimented with driving the robot to a specific location using a Representational State Transfer (REST) application programming interface (API) and proved to be successful in both cases [6]. Their results were promising, however this solution is not customizable and does not satisfy the specific needs of the company.

In a joint effort with Escio, a prototyped system solution with integrated support for delivering IoT data to their internal systems was discussed and drafted. The prototype had to be developed in a two-part process; Part one would be to create a Hypertext Transfer Protocol (HTTP) API using existing technologies for publishing messages to AWS. Part two would be to implement an API for a specific IoT protocol for automatic reporting of sensor data. This data would then be handled and used to control the AGV.

1.2 Project Definition

The initial problem was to develop a system for receiving and sending two-dimensional data to AWS cloud using an Hypertext Transfer Protocol (HTTP) API. The project scope was later extended. The extended phase is a separate prototype of a system that incorporates the use of IoT sensors and an AGV. Additionally, it should be able to use these two systems in conjunction with each other.

The goal of this project is to create a prototype of a unified system for IoT sensor data transmission and controlling an AGV based on this data. It should also be possible to use the web API to simulate data. AWS and IoT technologies are both central parts of the project, as well as researching and experimenting with different network protocols and AWS services.

Project Results

- Create a prototype API for sending/receiving user or sensor data for requesting an AGV to predefined locations.
- Produce a report that provides insight and research about the different technologies.

Project Impact

- Prove that such a system is feasible for use in different settings to increase effectiveness

1.3 Target Audience

The product and the bachelor thesis have their own target audience. The developed product is aimed at Escio and their employees, where as the thesis is mainly for academic purposes.

Product Audience

Target audience for the finished product are Escio's employees. The prototype will be primarily used by the company employees for the purpose of demonstrating the technology behind the prototype.

Thesis Audience

This report will be directed towards readers interested in the topic and/or has a wish to work with the IoT pipeline. The reader should have some technical knowledge about programming languages. Knowledge about IoT and cloud services is recommended but not required.

1.4 Project Organization

Every team member had equally shared responsibilities for development and writing of the thesis. In addition, each member had one or more roles to fulfill throughout the project.

1.4.1 Distribution of responsibilities

Eirik P. Masteholtet was chosen as project leader and had the main responsibility for designing the system architecture and overall project structure. As project leader, he makes sure that the project goes according to the plan and meets the requirements. He also has the authority to make major decisions, keeping the group's best interests in mind.

Bartosz Tracz was chosen to be project manager. As project manager his main responsibilities is things like keeping track of deadlines, coordination of the workflow between group members and report on the project progress.

Anders Isaksen was primarily responsible for all communication between the team, product owner (Terje Krogstad) and supervisor (Hao Wang). He also had responsibilities such as scheduling meetings and take care of all the permissions needed for the various services.

Hao Wang is an associate professor with a Ph.D in computer science and field of expertise in fields like IoT, Big Data and High Performance computing. His main functionality was to supervise us throughout the entire project.

Terje Krogstad is a senior developer at Escio. His main functionality was to provide us with the required equipment and devices. He also provided guidance in our software development process.

1.4.2 Rules and Procedures

The team has established some rules that needs to be followed. As a rule we decided to meet up at least twice a week and have a minimum of 30 work hours. Daily stand-ups will be held either in person or over the Discord communication service in order to keep each other up to date on the current state of the project.

If a team member does not show up to meetings that is organized outside of the daily standups without notifying the team in advance, he will face a penalty. The member will be required to work up the total hours lost in his spare time.

1.5 Constraints

The project consisted of some resources that had limitations on the scope. Criteria such as time, technology and legal constraints will set the framework of the project.

1.5.1 Time Constraints

We had two major deadlines for the project. The first was the feature freeze on the 11th. of April. After this date, no additional features were to be implemented, but the team could set aside some time for bug-fixing and code improvement. The second deadline was the thesis submission date, which was due to 20th. of May.

1.5.2 Financial Constraints

As research was a big part of this project, there were no strict financial constraints. We had an ongoing dialog with the company regarding hardware and equipment that were needed in our system. Server cost and computing power is financed by Escio.

1.5.3 Legal constraints

As the project scope got extended, a non-disclosure agreement (NDA) had to be signed to safeguard the company's interests. Due to the NDA, some of the more descriptive parts of the project has been left out or rewritten in such a way that protects the company's intellectual property.

1.5.4 Technical constraints

The Raspberry Pi is used as our software development platform. As a consequence, this solution have some hardware limitations such as memory and computing power.

1.6 Thesis Structure

Chapter 2: *Background*

This chapter provides the reader with the theory and background recommended for this thesis.

Chapter 3: *Requirements*

Presents a general overview of the client's requirements for functionality and quality.

Chapter 4: *Choice of Technology*

Provides information behind every choice that was made during the project. Covers different aspects like choice of programming language, IoT protocol an various frameworks.

Chapter 5: *Technical Design*

This chapter provides a high-level overview over the system architecture and a detailed overview over the sub-systems.

Chapter 6: *Implementation*

Describes the implementation process of the REST API as well as prototype of the sensor controlled AGV. This chapter provides information about the challenges that occurred, solutions to these problems as well as the achieved results.

Chapter 7: *Development Process*

Covers the development process and describes various development tools, testing, documentation and methodologies used during the project period.

Chapter 8: *Discussion*

This chapter describes the final results of the project. Topics like alternative choices, critique of the thesis and thoughts around project organization are also discussed.

Chapter 9: *Conclusion*

Chapter 9 concludes this thesis. It provides our contributions and describes whether we have achieved the goals for this thesis. This chapter also presents our thoughts on future work.

2 Background

To get the most out of this thesis, some knowledge of the following content is required. This chapter is intended to give the reader some theory about the topics covered in the subsequent sections.

2.1 IoT

IoT is a network of devices such as vehicles and home appliances that contain electronics, software, actuators, and connectivity which allows these things to connect, interact and exchange data[7]. "*The basic premise and goal of IoT is to 'connect the unconnected'*"[8]. In practice, this means connecting devices and objects that previously could not communicate with each other.

There are many ways to make IoT devices collaborate. This is done using different protocols that can communicate with each other over various network topologies. In this section we will describe most common IoT protocols and network connection methods.

2.1.1 MQTT Protocol

MQTT is a publish-subscribe messaging transport protocol. It is open and lightweight protocol, designed for easy implementation. These characteristics makes it ideal in many situations including constrained environments. It is commonly used for communication in Machine to Machine (M2M) and IoT contexts where a small code footprint is required and/or network bandwidth is at a premium [9][10][11].

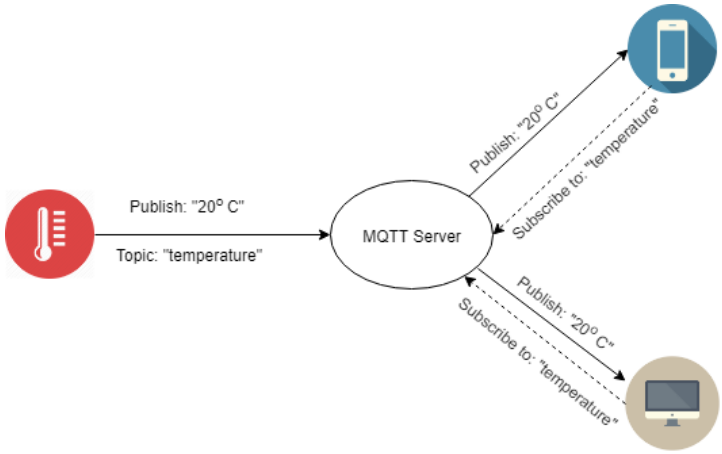


Figure 1: A representation of the pub/sub pattern where a client publish to two subscribing clients.

MQTT is based on the publish-subscribe (pub/sub) pattern where the senders of the message, called

publishers, sends messages to receivers, called subscribers as seen in Figure 1. The client that publishes a message is decoupled from other publishing clients or clients that receive the message [10]. The content of the message does not contain information about who the message is intended for and therefore do not know about the existence of the other clients. In order to publish or subscribe, the clients needs to establish a connection with an MQTT server, also known as an MQTT broker in the earlier versions. This protocol use a topic-based filtering system for messages. A server filters the incoming messages and distributes them to the clients that subscribe to topics that pertain to them, and therefore receive messages that are published about those topics [10][11].

2.1.2 Network Topologies

There are a number of different types of network topologies, including point-to-point, star, mesh, tree and more. A topology refers to the virtual layout of the interconnected devices on a computer network [12]. In this section we will be focusing on star and mesh topology, and describing differences between them. Figure 2 visualizes different connection patterns for these topologies.

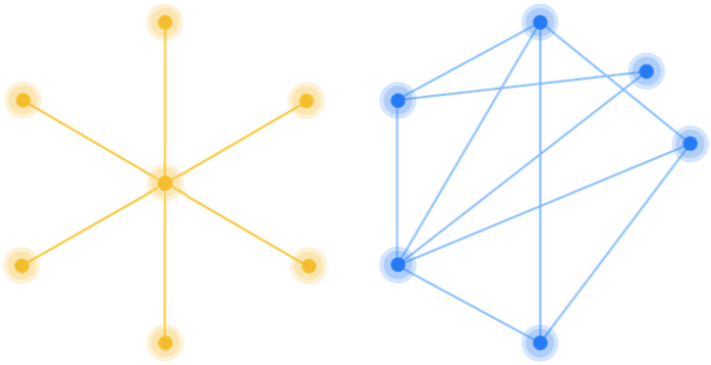


Figure 2: Star and Mesh Topologies [1]

Star topology is one of the most common network setups. In this configuration, every node connects to a central network device called hub or switch. The central network device acts as a server and the peripheral devices act as clients. All of the nodes on the network must be connected to one central device which handles all the traffic in the network. The network is robust in the sense that if one connection between a node and the hub fails, the other connections remain intact. However the primary disadvantage of the star topology is that the hub represents a single point of failure which means that if the central hub fails the entire network goes down [12][13].

A mesh network is a local network topology in which the infrastructure nodes connect directly, dynamically and non-hierarchically to as many other nodes as possible. The nodes cooperate with one another to efficiently route data from/to clients. Each node in the system acts as both a data source and a repeater. Information from a single node travels from node to node until the transmission reaches the destination. If the message can be sent directly between node A and B it is done so. A

message from node A to node C can be successfully delivered even if the two nodes are not within range, providing that a third node B can communicate with nodes A and C. If the preferred route is unavailable, the message originator will attempt other routes until a path is found to the C node. As a result, a mesh network can span much farther than the range of a single unit [12][14][15].

2.2 Cloud Computing

By definition, cloud computing is the practice of using a network of remote servers hosted on the Internet to store, manage, and process data, rather than a local server or a personal computer. It is a computing paradigm where a large pool of systems are connected in private or public networks. It provides dynamically scalable infrastructure for applications, infrastructure and storage services. [16][17]. This allows services to be rapidly provisioned and released with minimal management effort from the user side or service provider interaction [18].

There are multiple cloud providers that offers services depending on the client needs, for example AWS, Azure and Google. "*A cloud services platform such as AWS owns and maintains the network-connected hardware required for these application services, while you provision and use what you need via a web application*"[19].

2.3 API

Application Programming Interfaces (APIs), including libraries, frameworks, toolkits, and software development kits (SDKs), are used by virtually all code [20]. APIs enable the reuse of libraries and frameworks in software development [21] by providing a programmable interface to a service making it possible to use in conjugation with other services and programs.

Interaction and communication with cloud services is handled by cloud API's. Applications can integrate these API's to be able to move workload and logic into the cloud. Most cloud service providers has developed their own API's that users can utilize, and each has their own benefits and challenges. However, many of them provides interoperability, so that the user can use multiple API's without the fear of compatibility issues [22].

3 Requirements

In order to gain a better understanding of what is desired of the end product, a general overview of the client's requirements for functionality and quality was needed.

3.1 Functional Requirements

Although the assignment was very open-ended with the supervisor encouraging researching and choosing the technologies most suited for the job, Escio had some predefined requirements:

- The developed API had to be HTTP based and build upon already existing industry standards such as REST or GraphQL.
- The API have to be able to handle real-time data transmission.
- All validation and data routing forwarded to AWS will be handled by AWS IoT rules

3.1.1 Use case

The use case diagram shows the involved parties that will use the system and their associated actions.

Actors

The actors in the use case as seen in Figure 3 is described as follows:

- **User:** The person using the application and has access to most of its functionality.
- **AWS:** Sends back data to system
- **API:** Creates and configures the required files needed for the API to function properly.

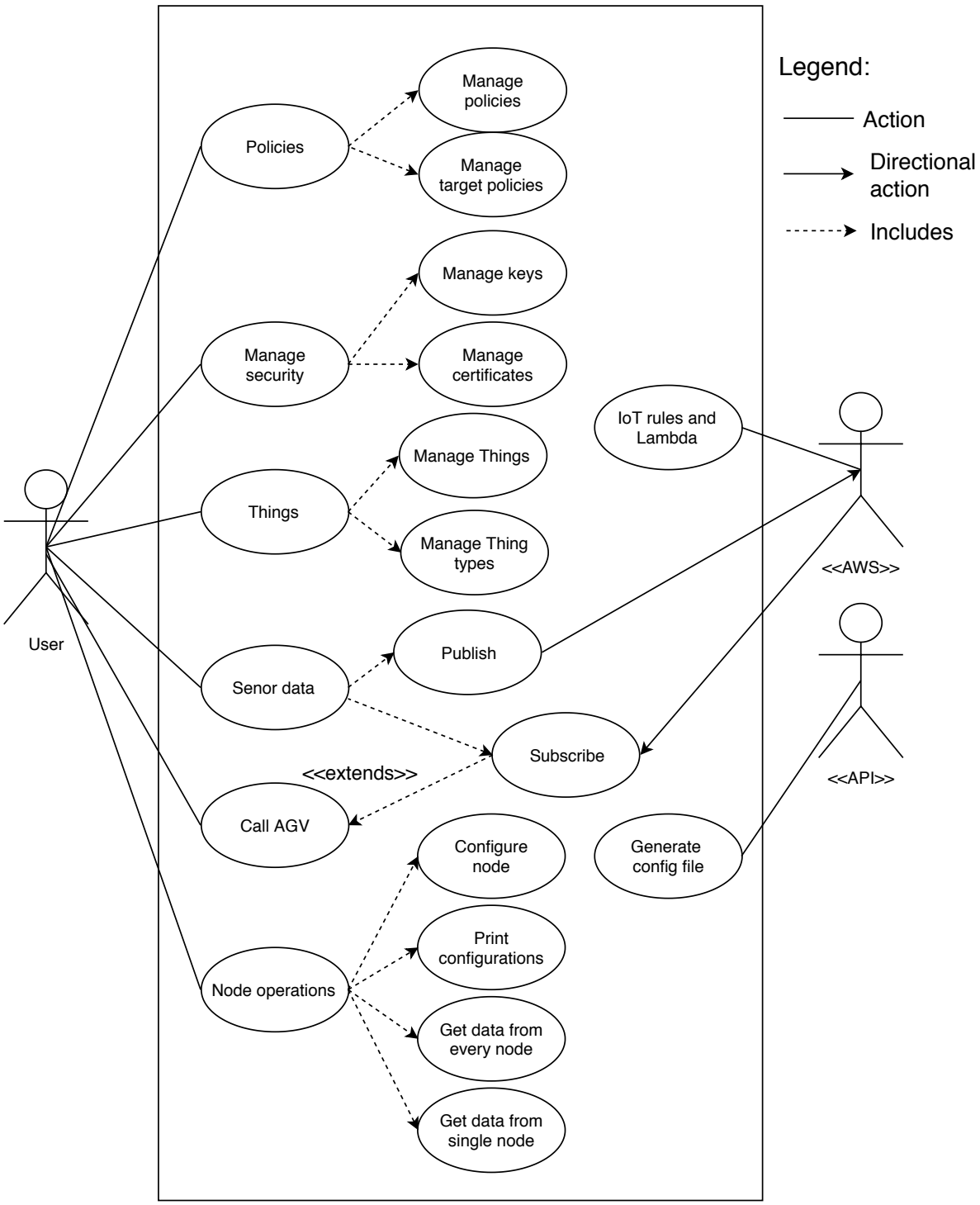


Figure 3: Use case diagram for the REST & Z-Wave API

3.1.2 High level use case

Use case	Policies
Actors	User
Purpose	Manage policies and target policies
Description	The user should be able to perform create, read, and delete operations as well as attach and detach policies on things. Policies allows the user to administer access to the AWS IoT data plane.

Use case	Manage security
Actors	User
Purpose	Manage keys and certificates for AWS communication
Description	Keys and certificates is a central part of the AWS pipeline. These allows the user to establish a (secure) TLS connection with the IoT core services. Keys are personal and each user has their own set of keys, the certificates can be found on the official AWS page and the user has to pick the right certificate for the job. An X.509 certificate is required for the TLS connection.

Use case	Things
Actors	User
Purpose	Manage things and thing types
Description	The user is able to perform create, read, update, delete operations on things and thing types

Use case	Node operations
Actors	User
Purpose	Handle operations and communication with the different nodes on the Z-Wave network.
Description	There are 4 different operations related to this use case: **Configure node** - see detailed use case in Section 3.1.3 **Print configurations** - prints the configurations of the node that the user is allowed to edit. The print contains the manufacturer name, config parameter id and current value. **Get data from every node** - Prompts all the connected nodes to report their name, temperature, UV, luminance and relative humidity. **Get data from single node** - Prompts a single nodes to report its name, temperature, UV, luminance and relative humidity

Use case	Sensor data
Actors	User
Purpose	Publish and receive sensor data from AWS Cloud
Description	Allow the user to: 1. publish sensor (or mock) data to AWS 2. Subscribe to incoming messages from AWS based on topics.

Use case	Call AGV
Actors	User
Purpose	Control the AGV
Description	The AGV can be controlled by received messages from either the user or from AWS.

Use case	IoT rules and Lambda
Actors	AWS
Purpose	Handles the received data based on created Rules
Description	AWS rules is a way to query data received on the server. These rules are created by a user, but handled by AWS. Lambda is configured to manage the data and control the logic.

Use case	Generate config file
Actors	API
Purpose	Generates a config file based on nodes in the network
Description	The first time the Z-Wave network is started, a config file is automatically generated. This config file contains a list of nodes and their configuration parameters.

3.1.3 Detailed use case

We opted for a more detailed description of the event flow regarding the configuration of a network node as this is a rather comprehensive and complex operation.

Use case	Configure node
Actors	User
Purpose	Edit the node settings such as report times and alarm thresholds
Preconditions	Node is paired to the network
Postconditions	The updated values can be viewed by prompting the node for its configurations
Main flow	1. The user runs the zwave.py script. 2. The script looks for existing configuration file. If it does not exist, it will be created with default values based on the sensor types. 3. The user creates a network object 4. Using the network object, the user will start the network 5. The network will prompt all connected nodes for status and returns network state 7 or 10. With a network level 7 or above, the user can perform node and network operations. 6. The user will make use of the *configure_node* method to update the node with desired values. The user will be referred to the node manual to examine valid configuration values. 7. The user will get a configuration completed message if the update was successful.
Alternative path	1. The script is unable to detect configuration file: The script stops and informs the user to provide a valid configuration path. 2. Network does not start: If the network is unable to start (caused by disconnected/faulty nodes) the script will exit with an error message. 3. Network never reaches level 10: This is caused by the network not being able to wake up sleeping nodes. All operations will function at level 7, but sleeping nodes will not be affected.

3.2 Technical Requirements

Being a prototype, the main concern is to see if this project is achievable. Therefore the team had the responsibility to chose the programming language, protocols, operating system and other technical aspects based on research and experience. On the other hand we were required to use the AWS cloud platform as this is the cloud platform they use.

3.2.1 Docker Support

Terje and the group agreed that the application should be able to run as a docker service. Due to the ease of use and portability of docker services meant that the application easily could be transferred and installed on different systems as needed.

3.2.2 AWS Requirements

As Escio uses AWS on a daily basis, one of the requirements were to use AWS services throughout the project. AWS has all the tools needed for IoT operations with their AWS IoT Core, as well as support for cloud computation and logic handling with AWS Lambda and IoT rules. There was a desire to have of the logic handling out to the cloud for purpose of scalability.

3.2.3 User interface

There were no specific UI requirements for the application. The team decided that the REST API should have a web-interface for usability.

3.3 Project Requirements

Escio required an agile software development process to be used during the project implementation. There was also a requirement to use Jira as this is what they used on a daily basis.

3.4 Security

As the application would only be a prototype and intended to run on a local network, there were no set requirements for security. However, if time allows it, the REST API should implement the Oauth2 authentication framework.

3.5 Equipment

Escio had an AGV of type MIR100 and the project was set to be centered around this unit. The AGV has an incorporated, proprietary REST API, meaning that calls to the AGV should happen with the use of this API.

3.6 Documentation

The documentation process will happen at two stages. Stage one is concerned with the development process and the other with regards to the coding implementation.

3.6.1 Documentation of the Development Process

All choices made during the development process should be documented. Additional requirements and functionality that appears during the course of the project should be handled by appropriate project planning tools.

3.6.2 Code Documentation

Code should be written and documented using the specific language requirements for code style. The project repository markdown file should give the user a good overview of the application and guidance on how to install and use the code. Documentation should be generated from source code with the appropriate tool within the chosen programming language.

4 Choice of Technology

A significant part of the project was associated with research. Choosing the right tools in order to accomplish the various challenges is a central part of prototyping. In this chapter we will describe various programming languages, development tools and frameworks we considered to use in our project. In addition, we will present each alternative and discuss its strengths and weaknesses and the motivation behind our choices.

4.1 Programming Language

The implementation process consisted of two parts. Both parts had their own unique requirements and approaches in regards to choice of language. The first part concerns the development of the web API, and the second part covers the implementation of a system for handling real time sensor data and control of an AGV unit. Overall the criteria for the programming language are as follows:

- Web framework support
- Works on Raspberry Pi
- Compatibility with IoT protocols
- AWS SDK support
- MQTT support

4.1.1 Candidates

Go is an open source project developed by a team at Google and many contributors from the open source community. We considered Go mostly because it is fast, has great support for multithreading and easy-to-read syntax. Go is a compiled language which means that it runs directly on underlying hardware. Go was built with multithreading in mind. It has goroutines instead of threads, which provides flexible memory management and avoids having to resort to mutex locking when sharing data structures. Go also has a strong standard library for web development [23][24]. Go is a young language compared to the other options and is still developing.

Node.js is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser. Node.js was mostly considered because of its great support for REST application development, and is well equipped to handle a large number of requests through streams. It is easily installed on for example a Raspberry Pi and has great support for sockets and MQTT operations[25].

Python is a high-level, interpreted and general-purpose dynamic programming language that focuses on code readability. We considered Python mostly for its ease of use, readability and great support with the web frameworks. The syntax in Python helps the programmers to do coding in fewer steps as

compared to other languages [26]. Python has a huge collection of various web frameworks, ranging from micro-frameworks, through asynchronous frameworks and ending with full-stack frameworks [27].

C/C++ both works great for applications running on devices where computing power usually is rather limited, such as a Raspberry Pi [28]. C works well because of its low-level access to the computer memory and does not require a lot of processing power. C++ is directly derived from the C language, which means it shares a lot of properties with C while also adding improvements and support for object oriented programming. Both C and C++ has low support for web frameworks, making development of a web API very challenging.

Java is one of the most widely used programming languages and a prominent choice for IoT engineers. It has been the backbone for many emerging IoT technologies [29]. One of the primary reasons for choosing Java would be its flexibility and versatility. It intends to let an application to be "written once, run everywhere". In later years it has been ported to the ARM Cortex family, which dramatically lowers the requirement for hardware needed to effectively run the Java VM [30]. Depending on the need of portability, Java might be a good choice.

4.1.2 Comparison

Table 1: Programming languages comparison table

Constraints	Go	Node.js	Java	C/C++	Python
Web framework support	Native	Great	Great	Low	Great
Raspberry Pi support	Good	Good	Good	Native	Native
MQTT client support	Low	Good	Great	Good	Good
IoT protocols support	Yes	Yes	Yes	Yes	Yes
Previous experience	Low	None	Good	Good	Great

Since we were developing a prototype and had a tight time schedule, choosing a language that would require a lot of time and effort to learn would not be a smart move. Every member had different experience with the mentioned languages, but Python, Java and C++ were the ones we all knew the best. Looking at Python's extensive library support and easy-to-use web frameworks, getting a prototype up and running would be much simpler than with e.g. C++ or Java.

At the end, Python was chosen based on several conditions. We were supposed to develop a prototype, which meant that focus on language performance and multithreading was low. Python has been out for decades and has great documentation, a huge code base and large community. Finally, the biggest reason behind our choice can be illustrated by Table 1. Python has the overall highest supported criteria and is the language that our team has the most experience with.

4.2 Web API design architecture

We were free to pick an API architecture of our choice. However, Escio suggested REST and Graph QL. Therefore, we took closer look on these two architectures.

4.2.1 REST

First introduced in 2000 by Roy Fielding, REST is an architectural paradigm providing a standard between computer systems on the web, making it easier for systems to communicate with each other. It is most commonly used to design APIs for web services [31]. REST is designed with a set of constraints [32]:

- **Client-Server** - In REST, the client and server are separate entities. This means that implementation and changes can independently be done on one part without interfering with the other.
- **Stateless** - Means that the server does not store any state about the client session on the server side. The client should provide all the information needed to handle the request. The server can service any client at any time.
- **Cacheable** - Clients can cache responses from the server. Responses has to implicitly, or explicitly, define themselves as cacheable. Well managed caching reduces the need for communication with the server, improving performance.
- **Uniform interface** - All services have a uniform interface and data is sent in a standardized form.

4.2.2 GraphQL

GraphQL was developed by Facebook in 2012, but was not made publicly available before 2015. Unlike REST, GraphQL is a data query and manipulation language for APIs. In GraphQL, all of the requests is sent to a single endpoint, which means that the developer does not need to plan out the uniform resource locators (URLs) in advance. Instead of the limitations of a uniform interface that REST provides, GraphQL can tailor requests (queries) to fetch exactly the data the user needs[33].

4.2.3 Comparison

The biggest difference between these architectural styles is:

1. In REST, accessible data is described as a linear list of endpoints, while in GraphQL it's a schema with relationships.
2. GraphQL has no form for web caching. GraphQL does not follow the HTTP specifications for caching, which REST does.application

Other differences and limitations can be visualized with the help of a Table 2.

Being familiar with REST and having the ability for caching, which was indeed described as an optional requirement for the project, this is what we decided to use.

Table 2: GraphQL vs REST comparison

Constraints	REST	GraphQL
Client-Server	YES	YES
Stateless	YES	YES
Cacheable	YES	NO
Layered System	YES	Only via HTTP POST
Code on Demand	YES	NO
Uniform Interface	YES	Only resource identifiers via node ID

4.3 Frameworks

After deciding on the programming language and architectural style, we had to make a choice between which framework to use. The two dominant web frameworks for Python is Flask and Django. Both have great documentation and are widely used. Following is a more in-depth review of these frameworks.

4.3.1 Flask

Flask is a micro web framework written in Python based on Werkzeug and Jinja 2. It is classified as a microframework, facilitating requests, routing, dispatching and returning HTTP responses [34]. Flask was mostly of interest because of its slim design and RESTful request dispatching.

Flask has no strict rule-set for project structure and scaling, but does provide recommendations and best-practices. This, for us, was one of the more negative aspects about the framework. By being so open ended and lenient on project structure, scaling a larger application could become an issue. However, this can be countered by using the RESTPlus extension and following the guide for scaling large projects.

4.3.2 Django

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. It is seen as one of the most productive frameworks for building the features needed by most medium to large web applications. Django favors an explicit approach, meaning, that the user has more a more direct control of the application behaviour. This approach requires more typing, but debugging and problem fixing becomes much easier [35].

In contrast to Flask, Django provides more functionality for creating a fully fledged application out of the box. Features like the object-relation mapping (ORM) make it compatible with most major databases. Furthermore it includes a ready-to-use admin panel for handling common administration tasks. However Django follows a strict project structure and wants you to stick to its modular philosophy.

4.3.3 Conclusion

Both are great choices and are very popular full-stack web frameworks. As setting up an caching service for logging data events and OAuth was an optional requirement, Django presented itself as

the best choice in our circumstance. With its included features the team did not need to spend time integrating different modules together.

4.4 Web Server

A Web server is responsible for receiving HTTP requests and forward a response back to the user. Django offers a lightweight development server. As the name suggest, it should only be used while developing and never in a production environment. Therefore a production server is needed.

4.4.1 Nginx

Nginx is an increasingly popular web server. It has grown in popularity since its release due to its light-weight resource utilization and its ability to scale easily on minimal hardware [36]. Nginx is about 2.5 faster than Apache for serving static content, and for dynamic content the performance difference is quite slim [37]. The web server also supports some more advanced features such as reverse proxy, load balancer and an HTTP cache.

4.4.2 Apache

Apache has been the most popular server since 1996 [38]. As a result of this, it is well documented and provide a substantial amount of integrated support and modules. It is a component of the well known LAMP (Linux, Apache, MySQL, PHP) stack which is still very popular.

4.4.3 Conclusion

The team members has previous experience with Nginx, and with its light-weight resource utilization made it the right choice for development on a Raspberry Pi.

4.5 SDKs and AWS connection helpers

Being an AWS oriented project, integration with their services using an SDK would greatly speed up the development process. AWS has developed an SDK for Python called Boto3, which enables developers to create, configure, and manage AWS services. This means that Boto3 will handle the authentication and data passthrough from local server to AWS.

An alternative to the Boto3 SDK would be to use direct calls to AWS services. AWS has a credentials provider that allows the use of the built-in X.509 certificate as the unique device identity to authenticate AWS requests. This eliminates the requirement to store access keys locally, but the provided token is only temporary and is somewhat limiting.

Boto3 offers all the functionality needed for a fast and efficient application setup. The application will be a self-contained system, so access keys can be handled locally on the OS. Access keys can also be added as a linux environmental variable, which means that different users accessing the same system can be limited by a system administrator and/or AWS IAM.

4.6 IoT protocols

There are currently plenty of various IoT protocols available on the market. Each protocol was designed with a specific use case in mind and provides its own set of features. In this section we will describe potential candidates as well as our choice for the IoT protocol for this project.

4.6.1 Zigbee

Zigbee is a low-cost, low-power, wireless mesh network (2.1.2) standard targeted at battery-powered devices in wireless control- and monitoring applications [39]. ZigBee uses the IEEE 802.15.4 specification for the lower medium access control (MAC) and physical (PHY) layers as its foundation. The protocol standard has two PHY layers that operates in two separate frequency ranges: 868/915MHz and 2.4GHz [40]. Data transmission rates vary from 20kb/s on the 868/915MHz band, to 250kb/s on the 2.4GHz band. Its low power consumption limits data transmission distances from 10 to 100 meters, depending on power output and environmental characteristics [41].

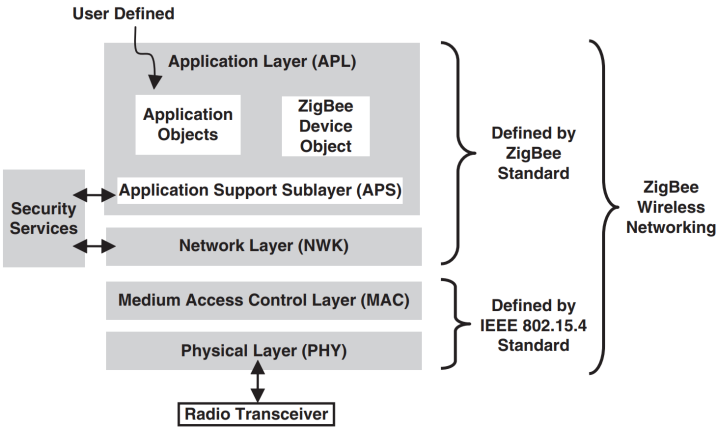


Figure 4: ZigBee Wireless Networking Protocol Layers [2, p. 5]

ZigBee is a specification of a network layer only. Application layer specifications were added later, but never made mandatory. This lead to a broad variety of ZigBee implementations coexisting on the market. All these versions claim to be ZigBee, but none of them is interoperable with each other. As the National Testing Service organisation states, one of the top five reasons for failures in ZigBee deployments is lack of interoperability between units [42]. Apart from the interoperability problem, Zigbee also faces additional challenges such as interference problems caused by the use of the congested 2.4 GHz frequency band [43].

4.6.2 Thread

Thread is an IPv6-based, low-power mesh networking technology for IoT products, intended to be secure and future-proof. With the joint effort of over 50 companies, Thread is an attempt at standardizing a protocol running on the 6LoWPAN to enable home automation [44]. The 6LoWPAN standard enable the efficient use of IPv6 over low-power wireless networks on simple embedded devices. The 6LoWPAN group has defined encapsulation and header compression mechanisms that allow IPv6

packets to be sent and received over IEEE 802.15.4 based networks [45]. The motive behind creating an IPv6 based protocol was the limited number of unique addresses that previous iteration of the protocol (IPv4) supported. IPv6 utilizes 128-bit addresses instead of the 32-bit adresses used in IPv4. This large adress space ensures availability of unique addresses for all nodes in any practical growth-rate scenario. Since the protocol is IP based, a wireless node that implements 6LoWPAN can be accessed and managed similarly to any other IP device [2]. Figure 5 shows the structure of the 6LoWPAN network architecture.

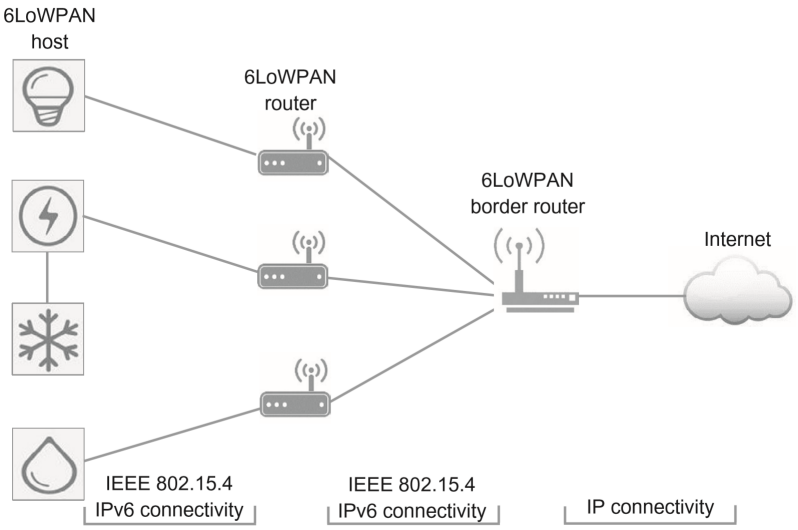


Figure 5: 6LoWPAN network Architecture [3]

4.6.3 Z-Wave

Z-Wave is a wireless communication protocol used primarily for home automation. Z-Wave is designed to provide reliable, low-latency transmission of small data packets at data rates up to 100kb/s [46]. Contrarily to the broadly known wireless network protocol (Wi-Fi), which sends data directly from A to B, the Z-Wave protocol uses a source-routed mesh network (see section 2.1.2) architecture to transmit data. Source routing allows a sender of a packet to partially or completely specify the route the packet takes through the network. Z-Wave uses the unlicensed industrial, scientific, and medical band that operates at the frequency between 865MHz and 926MHz, depending on the country regulations [14]. This frequency spectre makes is perfect as a home automation tool for indoor use, as it avoids interference with the crowded 2.4GHz band such as Wi-Fi and Bluetooth. Figure 6 shows how the Z-Wave protocol operates with the mesh network.

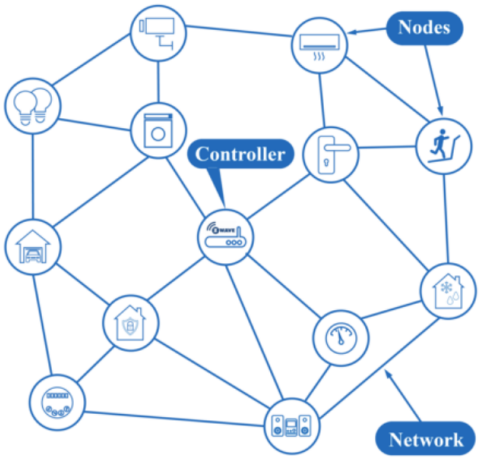


Figure 6: Z-Wave Mesh Network [4]

The Z-Wave protocol was privately owned and kept as a proprietary standard until 2016[47]. This had both its positive as well as negative sides. On the positive side every device manufactured had to fulfill certain standards and each newly made device had to be interoperable with existing devices. On the negative side vendors have been forced join the Z-Wave Alliance to gain access to developer resources. In addition, the cost of producing a Z-Wave device was higher due to the licensing fees. Since 2016, it has slowly made its way to the open source market. [47][14]

4.6.4 Comparison

Table 3 shows the comparison of these protocols.

Table 3: Comparison of the IoT protocols

Protocol	Network Topology	Data Rates	Range	Frequency Band	Max Devices	Interoperability
Z-Wave	Mesh	20kb/s - 100kb/s	40-100m	~900MHz	232	Fully
ZigBee	Star, Tree, Mesh	20kb/s on ~900MHz 250kb/s on 2.4GHz	10-100m	~900MHz / 2.4GHz	65000	Partly
Thread	Thread(Mesh) 6LoWPAN(Many)	20kb/s on ~900MHz 250kb/s on 2.4GHz	10-100m	~900MHz / 2.4GHz	2^{64}	Partly

We presented our findings to Terje and had a discussion regarding each presented alternative. Every protocol shares plenty of features, like source-routed mesh network architecture, open-source, plenty of supported vendors and devices. As the Z-Wave protocol works on the lower spectrum of the frequency band and all devices are fully interoperable, it presented itself as the most appealing alternative.

4.7 Choice of IoT Framework

There existing a lot of applications to help with the network communication of IoT devices. Many of these have a broader purpose and supports multiple IoT protocols in the same package. Some of these are open-source alternatives such as Home Assistant (HASS), openHAB and Domoticz. The frameworks are usually optimized to run on devices such as a raspberry pi.

Terje had some previous knowledge with HASS as he used this on a weekly basis. HASS provides a clean and user friendly web-interface where users easily can administer their connected devices. HASS is a great way to get familiar with IoT technologies as it serves as a way to abstractify the more complicated layers of IoT such a network protocols and data transfer. The main issue with HASS is that it tries to be a one-for-all solution for multiple IoT units. Using only one IoT protocol and a selected set of operations, HASS is definitely excessive for our use.

Escio desired a more single purpose application, so we started looking at various light-weight IoT frameworks. Knowing what type of protocol and programming language we would be using, we found several open source frameworks [48][49][50]. We chose the python-openzwave framework which is a python wrapper for the openzwave C++ library. The python wrapper was not fully implemented but had just enough functionality to create a fully customizable, lightweight Z-Wave API which is what the company desired.

5 Technical Design

In this chapter we will discuss our system architecture. Since our systems is disjoint we will have a high-level description of the technical solution, but also a more in-depth look at the two two systems separably. These solutions was designed with the goal of addressing all requirements described in Section 3.2.

5.1 System Overview

Most of our system components run separably as their own Docker service as seen in Figure 7. This way, the system could comfortably be ported to other platforms and operating systems. A Raspberry Pi was used as the local environment to host the different services.

5.2 Django application

As the REST API consist of an HTTP server and a WSGI server our system will follow the layered architecture pattern. Nginx represents our presentation tier and is responsible for displaying the user and serving static content. Requests that demands dynamically content is passed on to the application server which is Gunicorn. This represents the application layer and is responsible for detailed processing. The data will be sent and stored on the AWS platform which serves as our data tier as illustrated in Figure 8.

This architectural pattern allows for functionality to be organized and separated into layers. The benefits of this is that each layer can be changed independently and makes the system more portable. One disadvantage with this is that it may introduce unnecessary code complexity for simple projects [51].

- **Client** - A user or application sending requests and receives data to/from the web server either via the implemented graphical interface using Swagger or using a command line (e.g. *curl*).
- **Nginx** - Used as the HTTP web server and handles the static resources.
- **Gunicorn** - Serves as a Web Server Gateway Interface (WSGI). WSGI forwards the requests from the web server and invokes the django REST API. It also serves as a middleware to handle the load balancing.
- **Django REST** - Process dynamic HTTP requests. Uses the Boto3 SDK for communication with the AWS IoT core services. Boto3 is also used to handle custom responses from AWS.
- **AWS cloud** - Stores and handles the IoT core data.

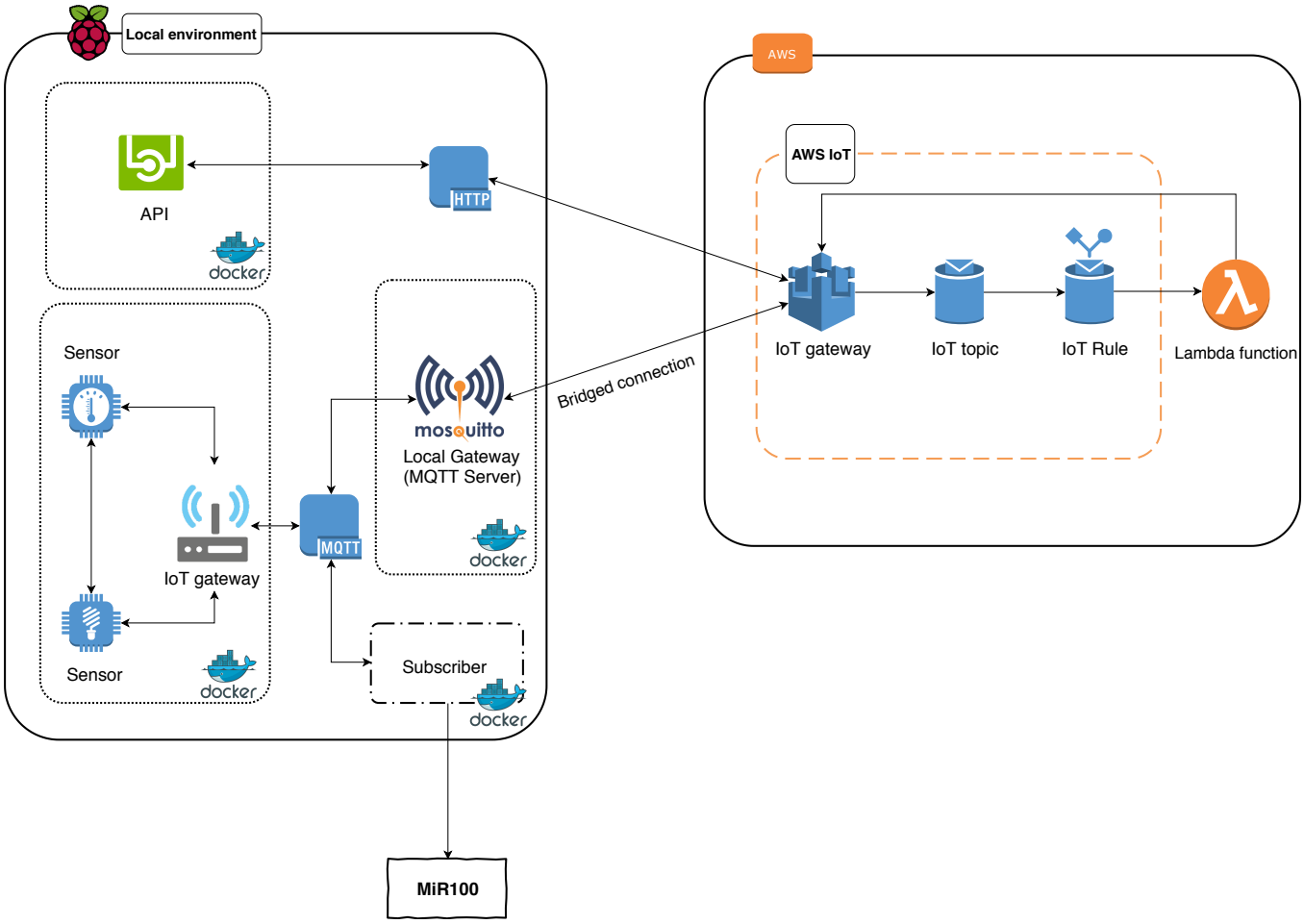


Figure 7: Complete system architecture

5.3 MQTT

MQTT follows the pub/sub pattern as described in Section 2.1.1. The communication process for moving the AGV is illustrated in Figure 9 and should follow the sequence in this specified order:

1. A controller connects to the broker and subscribes to a specific topic with an SUBSCRIBE message. The broker will send a subscribe acknowledgement (SUBACK) message back.
2. A sensor will try to connect with the broker as client. If the client fails to connect by providing a wrong username and password the connection would be refused, closing the network connection. Otherwise the broker will send back an acknowledge connection (CONNACK) Packet.
3. The client will publish its data to a topic and then receive a publish acknowledgement (PUBACK) packet from the broker.
4. The broker will forward the message to the AWS IoT's broker through the bridged connection and receive a PUBACK.

5. The data will trigger an AWS IoT rule and requesting the AWS Lambda service.
6. AWS Lambda will publish a command to the topic the local controller is subscribed to over HTTP.
7. AwS IoT will publish this message back to our local broker through the bridged connection and receive a PUBACK.
8. The broker will publish this message to the controller and receive a PUBACK.
9. The controller will request the AGV to move to a specific location based on the message from AWS Lambda.

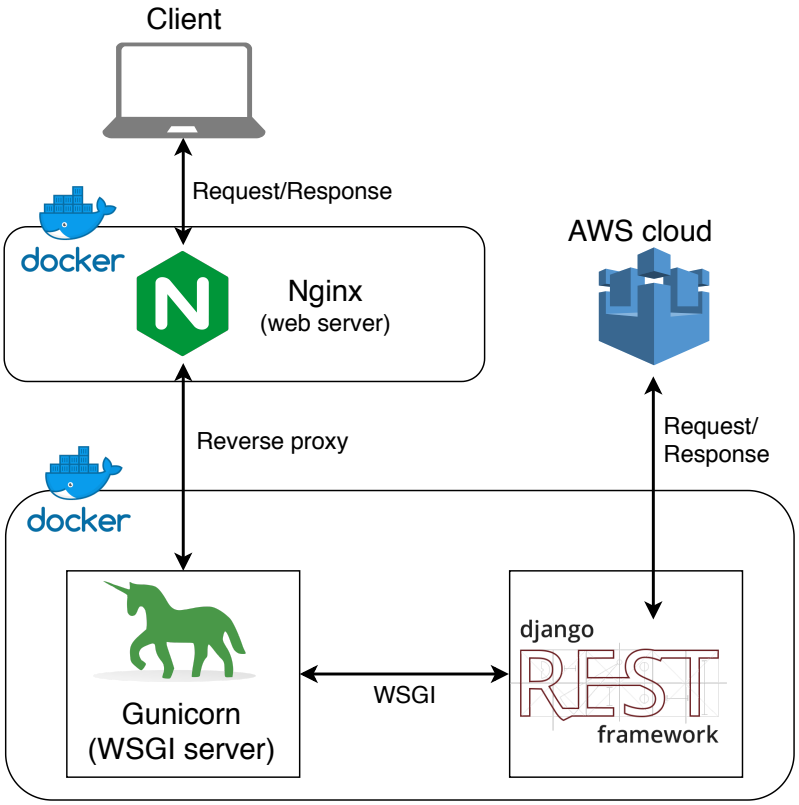


Figure 8: REST API architectural design

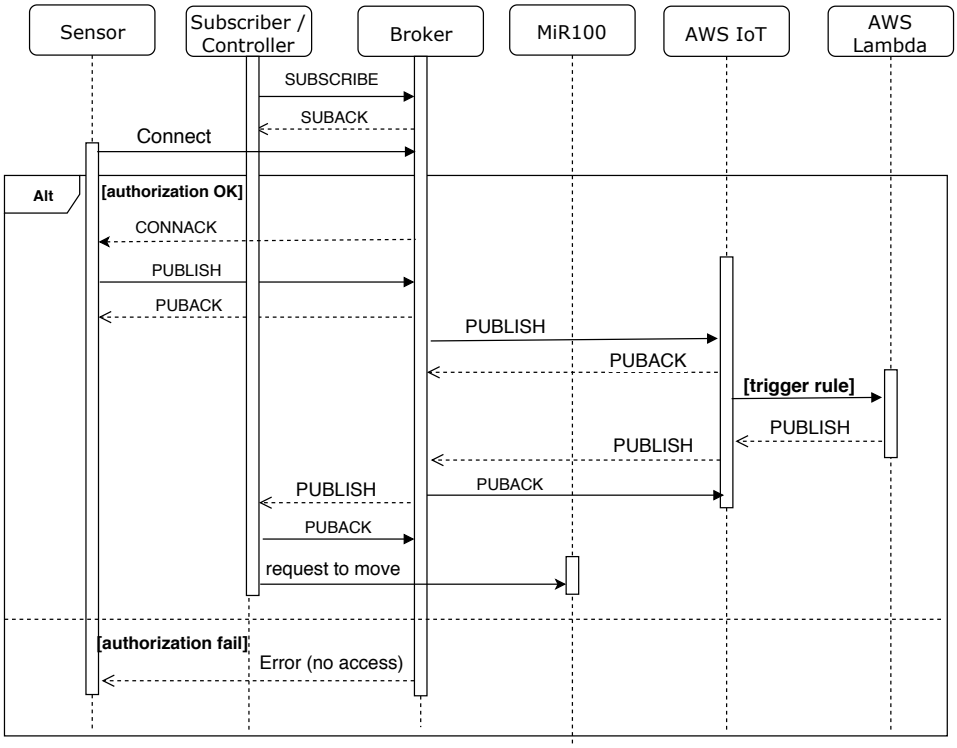


Figure 9: MQTT sequence diagram

6 Implementation

In this chapter we will describe the implementation process. It is divided into four parts: The REST API, Z-Wave API and how it collects the data, the MQTT connection between AWS and our system and finally how it all is integrated together.

6.1 REST API

The REST API was the first thing that was implemented as it was the first/initial requirement before the scope was extended. The API is developed with the help of the Django framework and Boto3 SDK and run as a docker service. The main purpose of this API was to receive multidimensional data from a user and forwarding it to AWS. Secondly we added additional functionality for handling all the administration surrounding IoT units. The REST actions were built with the help of the AWS IoT reference guide.

6.1.1 SDKs and third party libraries

For the development of the REST API, we used the AWS SDK for Python called Boto3. This SDK enabled us to create, configure and manage different AWS services. To get up and running with Boto3, we had to setup our credentials used to verify the connection to the different services. Although this could be done manually by creating required folders and copy-pasting the IAM credentials, we decided to use the AWS Command Line Interface (CLI) to manage this. AWS CLI automated the credentials process by running a *aws configure* bash command. The tool was also used for verifying the developed class methods as it included different IoT core functions.

6.1.2 Publishing data via REST API

The API allows the user to manually publish data to AWS with HTTP POST by either using the publish class method or via command line. Using the REST interface will allow the user to easily customize the URL and payload. For example, by navigating to the Swagger interface and looking at the documentation for publish as seen in Figure 10, we can see that the URL requires a topic and a QoS (Quality of Service). The message broker uses topics to route messages from publishing clients to subscribing clients. The QoS provides information about the package and can have a value from 0 - 2:

- **QoS level 0** - means a message is delivered zero or more times. A message might be delivered more than once. No puback is sent.
- **QoS level 1** - message is delivered at least once. A puback is sent back to the client when message is received.
- **QoS level 2** - handles message duplication. Message is received exactly once. AWS IoT does not, as of this date, support this QoS.

Figure 10: Swagger URL helper

If the user wants to publish data to topic */sensor/hallway* with a qos of 1, the URL would look like in Listing 6.1.

Listing 6.1: Topic URL example

```
http://localhost:8000/iot/topics/%2Fsensor%2Fhallway=1
```

By visiting this URL, the user will be greeted with the Django REST interface as demonstrated in Figure 11. This lets the user construct a payload by providing a simple textbox. The user can chose different media types to send, but we are most interested in the *application/JSON* media type. By filling out the textfield with desired JSON formatted data and clicking the POST button, the data will be sent to AWS IoT's message broker.

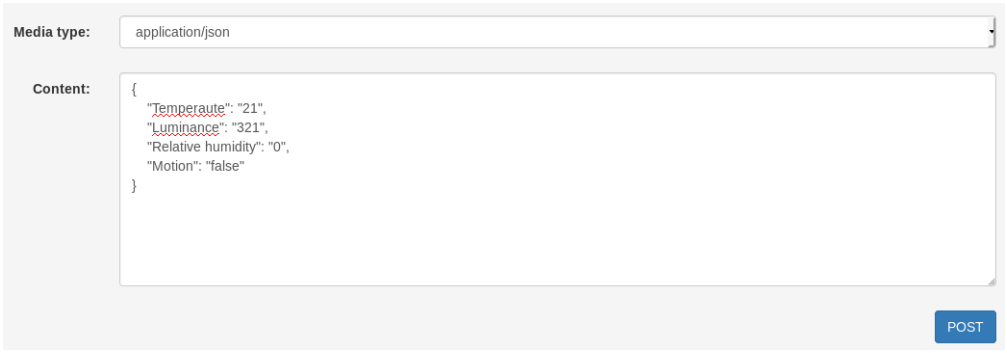


Figure 11: Django REST graphical interface

AWS IoT core has built-in functionality for testing publish/subscribe activity. By visiting the online AWS management console and subscribing to */sensor/hallway*, we can observe that the data was received successfully (Figure 12).

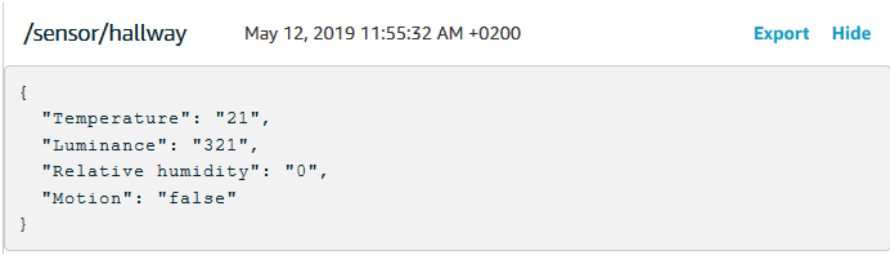


Figure 12: Received data displayed in the AWS management console

6.1.3 IoT core operations

Expanding on the minimum functionality, we decided to implement support for the most used IoT core operations. This was done in order to provide convenient functionality to the project and add support for automation as discussed in Section 9.1.

Following the Django REST standard, each endpoint/resource should have its own class/method inside the view function. This class contains all the desired protocols such as GET, PUT, DELETE and more. AWS IoT has over 160 core operations that can be implemented, but we decided to only implement the most useful operations for this project.

Thing - As described by AWS: "a representation of a specific device or logical entity". Usually, for each IoT device you own, you'd want to create a thing associated with that device. Although not required, adding these would allow for easier management (Listing 6.2).

Listing 6.2: Thing operations

```
URI: /iot/things HTTP/1.1
Methods: [GET]

URI: /iot/things/<thing_name> HTTP/1.1
Methods: [GET, PUT, POST, DELETE]
Content-type: application/json
```

Thing types - allows for storage of description and configuration information that is common to all things associated with the same thing type. Adding thing types would allow for multiple things to share the same set of attributes (Listing 6.3).

Listing 6.3: Thing type operations

```
URI: /iot/thing-types HTTP/1.1
Methods: [GET]

PATH: /iot/thing-types/<thing_type_name> HTTP/1.1
Methods: [GET, PUT, POST, DELETE]
Content-type: application/json
```

Policies - Policies are used to authorize different devices to perform AWS IoT operations, most notably subscribing or publishing to MQTT topics. This function allows the user to perform create, read, update and delete operations on policies (Listing 6.4).

Listing 6.4: Policy operations

```
URI: /iot/policies
Methods: [GET]

URI: /iot/policies/policy_name
Methods: [GET, PUT, POST, DELETE]
Content-type: application/json
```

Target policies - This method allows the user to list all the things attached to a specific policy, attach a new one or detach an existing policy (Listing 6.5).

Listing 6.5: Target policy operations

```
URI: iot/target-policies/<policy_name>
Methods: [GET, PUT, POST, DELETE]
Content-type: application/json
```

Certificates - IoT communication with AWS is protected using x.509 certificates. In order to allow the device to speak with the cloud, certificates has to be created and activated (Listing 6.6).

Listing 6.6: Certificate operations

```
URI: iot/certificates/<certificate_id>
Methods: [PATCH, DELETE]
Content-type: application/json
```

6.2 Z-Wave

Throughout the Z-Wave implementation phase we were faced with various challenges. Many of these challenges occurred due to our lack of previous knowledge about the Z-Wave technology, but some were a direct consequence of the python-openzwave wrapper. In this section we will be describing what kind of problems and challenges we faced as well as how we handled them. At the end of this section we will be presenting the results that we managed to accomplish.

6.2.1 Challenges and Problems

One of the major inconveniences of the python-openzwave wrapper was the documentation. Many of the methods lacked documentation and some methods that were documented was not fully implemented. This led to a lot of trial and error and reverse engineering to figure out how to use some of the functionality. Sometimes we had to look at the C++ source code to figure out how things were implemented.

Node configuration also proved to be more challenging than initially expected. Each node will generate an associated section in the main Z-Wave config file. For this section to be generated, the node has to be linked to its manufacturer. The configuration process is controlled by an extensible markup language (XML) file that works as a look-up table between the node id and manufacturer name, see Figure 13 for a clearer overview. Additionally, we struggled to configure one of the specific nodes as some necessary functionality was commented out.

The problems and challenges can be summarized as follows:

- Bad documentation and occasional lack of implementation
- No guidance in any form as to where to start and how to proceed onward
- Need for manual addition of manufacturer information for specific devices
- The need for reverse engineering of the software in order to understand it

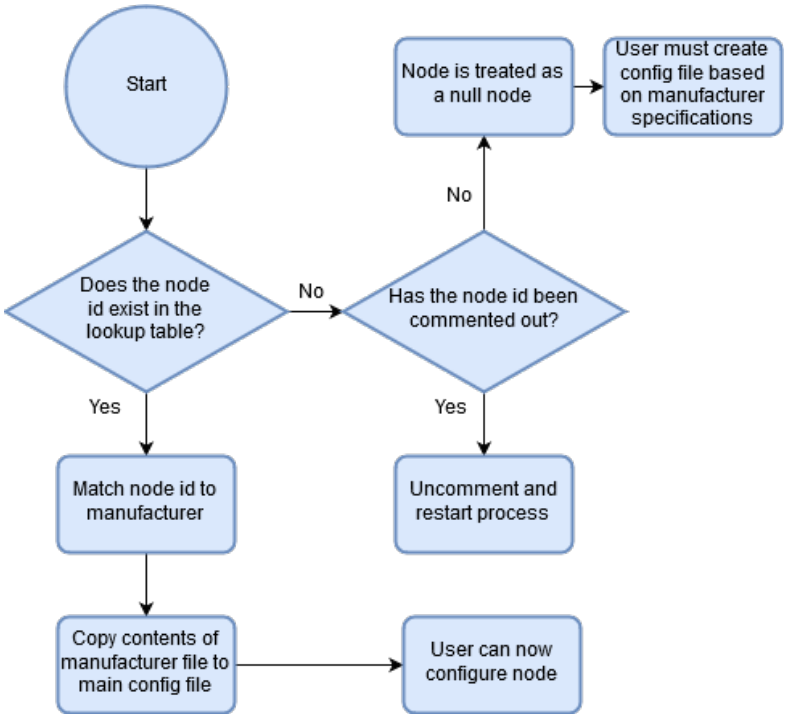


Figure 13: Adding node configuration to the Z-Wave config file

6.2.2 Solutions

We decided that the best way to start would be to do more research about the wrapper as well as read the documentation carefully. We cloned the Github repository for the wrapper for a more thorough analysis. We encountered a "test" folder in the repository and within the folder a file named "api_demo.py". We ran the file, hoping it would give us more insight, but all it did was to output network logs which did not prove to be useful. We examined the Python code inside the file and together with newly acquired knowledge from reading the documentation, slowly began to get a better understanding of the system. We started experimenting with the code while reading the function descriptions in the documentation and got to the point of successfully starting the Z-Wave network.

Our next objective was to establish a successful connection with the devices and receive data from the sensors. After a few days and a lot of trial and error we still did not get any sensor values from

the devices. This led us to question if the Z-Stick and the sensor devices were functioning correctly. In order to verify our skepticism we downloaded and installed Home Assistant in an attempt to control the network. We managed to get the sensor values from all of the connected devices, proving that the network was indeed working correctly. Swapping from Home Assistant back to our implementation and re-running the network, we suddenly started to receive data from the sensors. This gave us motivation to resume our research. We started to explore the ways Home Assistant handles the sensors, and since it is an open-source software, we analyzed its source code and functionality, like we did earlier with the Python wrapper. We discovered that HASS produced an XML configuration file that was used to store meta-data about the sensors and cache values. We included the configuration file in our code and could confirm that this file were responsible for the correct sensor behavior. At this stage we had managed to successfully receive the sensor values from the devices, only using our API. However, for some reason the values were static and did not update.

The next objective was to find out why the devices were not updating their sensor values automatically. After spending much time researching, we eventually found what we were looking for in the device manual. We discovered that the sensors had a default rate of updating their values set to 1 hour in order to maximize the battery life of the devices. Simultaneously, we discovered that we could force sensors to update their values by pressing a button on given device. Values were sent successfully through the network verifying that our implementation functioned correctly.

The next step was to find a way to configure the sensors in order to increase the poll frequency. We decided to use Home Assistant for this step as the Z-Wave wrapper had no documentation nor any guidance on how to configure a device. The official Home Assistant web site as well as their YouTube channel had plenty of information about the setup and configuration process. We managed to correctly configure the devices and get them to automatically update their sensor values once every five seconds. After confirming that the devices were configured properly, and were able to send updated data more frequently, the next step was to implement this functionality as a part of our API.

The process of configuring the devices without the need of Home Assistant, was the longest and most tedious. As previously stated, the documentation and resources is scarce. The python-openzwave documentation did contain some various configure functions, but when tested, the configurations did not take place. Again, we used to Home Assistant to see how it was configuring the devices and consorted to reverse engineering in an attempt to recreate this process. The reverse engineering as well as the documentation provided by Home Assistant led us in the right direction. We found that there are multiple configuration files that we need to take under consideration. We tried changing some values in the configuration files but did not get the desired results. Fortunately, as we were searching through the web, we found that someone had posted a similar problem in the python-openzwave GitHub issue thread [52]. The answer was provided by David Vassallo, who was successful in figuring out the whole configuration process [53]. Carefully following his response, we were able to get the desired results of updating the sensor values more frequently. This was a huge milestone as it allowed us to encapsulate every useful functionality within our API.

6.2.3 Results

The result is a fully functioning, self contained API that handles and manages data from IoT devices on a Z-Wave network. The Z-Wave API class consists of 6 methods:

start_network() - This is the first method that must be executed after initializing a Z-Wave object. This method is the building block for the whole network and has to be executed in order to run other methods. It is responsible for initializing the mesh network and the inclusion of all the connected nodes.

get_every_sensor_data() - This method returns sensor data from every available device on the network. If the device is paired but not turned on when the method is called, the method will return cached data from last successful poll. The return format is a Python dictionary with the sensor name and its value for each device. If the user wish to retrieve sensor data from a specific device the user can call the **get_sensor_data()** method.

get_sensor_values() - Returns a dictionary with available sensor values and their description for each device. The values uniquely identifies each attribute for a node.

print_configs() - Prints information about available configurations that can be made for a specific device. It is recommended that this method is called before configuring a node as it presents the information about the parameters and their associated values that the user can configure.

configure_node() - This method is mainly responsible for updating the polling intervals or changing alarm thresholds for a device sensor.

6.3 AWS Communication

There are many things to keep in mind for handling the communication with the AWS cloud. What kind of protocols to use, brokers and security are some of the key elements to a functional AWS workflow.

6.3.1 MQTT Server

AWS IoT provides multiple ways to communicate with IoT devices, for instance MQTT and HTTP. Our IoT devices have network capabilities but are not supported by AWS IoT. Therefore, we needed to build a gateway that serves and translates the IoT device protocol to the AWS IoT protocol. The project description mentioned that MQTT was something that micro-controllers and system on chip could use to communicate with AWS, so our team spent some time researching this technology and how it could be incorporated into our project.

Challenges and Problems

Our first concern was to be able to create a connection with AWS. The connection needed to be bidirectional for allowing communication between the AWS platform and our system. In order to

establish a connection an authentication process was required by the AWS before successfully connecting. This was the main challenge during this implementation part. As a consequence of MQTT working on top of the TCP/IP protocol it does not use encrypted communication by default. AWS uses various authentication mechanisms depending on the connection protocol from the user.

During our development, we encountered two different authentication processes. The first one was with the Transport Layer Security (TLS) protocol and the second was the Signature Version 4 over HTTP. Both adds security at the transportation level of the message delivery. The TLS protocol depends on a X.509 certificate as mentioned in Section 3.1.2. A second resource that is needed is the root certificate authority (CA). CAs issue certificates to specific domains, so when the client presents a certificate that is issued by a trusted CA, AWS knows it is safe to make the connection [54]. One issue we ran into in the beginning was that we downloaded the wrong root CA from AWS unknowingly, causing it to refuse the connection from our client.

The Signature Version 4 authentication on the other hand does not utilize certificates. This process relies on the users security credentials which consist of two parts, an access key ID and a secret access key which is also known as the users IAM credentials. In addition, it also requires the root CA like the first alternative.

The process above provides a direct connection with AWS, however we also considered an alternative for constrained environments. There might be devices in a industrial environment that cannot be exposed directly to the internet. As a consequence of this, a local server that handles the communication between AWS and local IoT devices would be needed. With no previous experience we did not know if this was a possible.

Solutions

When researching programming languages that had the required criteria for implementing this system, we found Paho. Paho is an open-source client implementations of MQTT provided by the Eclipse Foundation. It allows the user to connect to a server by specifying the URL and port. Considering AWS does not allow for a raw TCP/IP connection we needed to specify that the connection was going to utilize the TLS protocol. As mentioned above this require an X.509 certificate, a root CA and a private key. The certificate could be downloaded through the AWS CLI or through our REST API. The public and private key is issued together with the certificate in both instances, however the private key will only be issued once and needed to be stored in a secure location. The root CA can be downloaded from the AWS home page, however the correct version needs to be selected. The last phase before connecting was specifying the path of the downloaded resources, our AWS endpoint and the port. The endpoint can be found in the settings panel inside the AWS IoT service site. Finally the port 8883 is specified since it is the standardized port for a secured MQTT connection [55][56]. As specified earlier, we initially downloaded the wrong root CA version which resulted in an unsuccessful connection.

With a failed attempt we looked at other options. Short after, we found AWS's own IoT SDK which is

built upon Paho. When comparing the two there is mainly one key difference, specifically its option to use another communication protocol. In contrast to Paho, the IoT SDK can also access the AWS IoT platform through MQTT over the WebSocket protocol. This protocol uses the Signature Version 4 authentication and the user needs IAM credentials and a root CA as stated above.

When initializing the client the SDK require the user to specify the location these files. There are two ways of doing this. The first option would be to hard-code the credentials in a custom configuration file and specify its path in a method parameter within the SDK. This is not recommended as it will expose the credentials in plain text making it easier to discover, and increasing the risk of attackers retrieving sensitive information. The second approach is to export these variables as environment variables. The SDK will search for these specific environment variables for credentials if no custom configuration is given: *AWS_ACCESS_KEY_ID* and *AWS_SECRET_ACCESS_KEY*. Unlike the previous attempt where we had to select our own root CA, the SDK's GitHub page provided a direct link to the correct version resulting in a successful connection.

With a successful result we wanted to explore the idea of a constrained environment. The Eclipse Foundation offer multiple open source projects, including Paho. With some research we came across one of their other projects called Mosquitto which is an open source message broker.

Through research we found that the Mosquitto broker have a feature called bridging. This provides the user with a way to connect two or more brokers together with some configuration. We found a blog post on AWS by Michael Garcia that describes the necessary steps to bridge our local Mosquitto MQTT broker to AWS IoT [57]. The connection protocol between them is TLS and the required resources for this is described earlier in this section. The user have the option to specify which topics that are bridged between the servers.

Results

Our first successful result is illustrated by Figure 14. The devices are directly communicating with the AWS IoT message broker without any intermediary broker. This solution requires that the device have the ability to connect to the cloud.

Our second successful result is shown in the system architecture (Figure 7). The result consists of local IoT devices connected to our local MQTT server that handles the communication between these devices and AWS IoT. However this implementation was not fully completed and have some issues that needs to be handled. See Section 8.1.2 for more information about these issues.

6.3.2 AWS Lambda

AWS IoT offers the ability to create rules based on the MQTT topic stream. These IoT rules allow the user to extract data from an MQTT message and send it to another AWS service. In this project the AWS system needs to publish a message back to our system when the content of the incoming message hits a certain threshold. The only service capable of this action is AWS Lambda. Lambda is a service described as "*a serverless compute service that runs your code in response to events and*

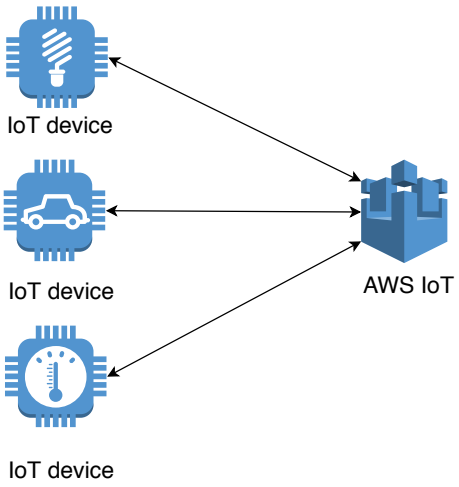


Figure 14: First successful result with IoT devices connecting directly to AWS IoT.

automatically manages the underlying compute resources for you" by the AWS team [58].

Challenges and Problems

There were uncertainties in the beginning surrounding how the Lambda function should publish an MQTT message back to our server given no prior experience with this service. One of the initial questions was how Python packages could be imported given no easy tools right out of the box. As a result of this there were no apparent solution on how to include an MQTT client library to the Lambda functions. Continuing on the problem with regards to publishing a new message through the bridged connection, some permissions were needed to be given the functions.

Solutions

Initially we considered the use of either the Paho client or AWS's own IoT Device SDK for publishing an MQTT message. First attempt was downloading the module to a zip folder and upload it to the function's deployment package. However, as noted by fengsongAWS, a contributor to the AWS SDK, these connections is designed to be long lived. As he states in a Github issue, "*[...] Lambda function is serverless and not designed for long lived connection, AWS SDK would be more suitable to be used in Lambda function*" *[59]*. With this helpful information we found out that Lambda includes the Boto3 SDK as part of the execution environment that can publish a message through the HTTP protocol.

For publishing to the AWS IoT some permissions were needed. This is achieved through a Lambda execution role giving the functions a "*iot*":*Publish*" permission to do an HTTP publish. With inadequate authorization to set these permission ourselves an email exchange between Escio and us was needed. As neither of us had experience in this area some trial and error was needed before the permissions was set correctly.

Results

AWS Lambda's role is to respond back to our system when an AWS IoT rule is triggered. With the SDK as a part of its execution environment we were able to create a function that could publish an MQTT message over HTTP as seen in Figure 15. Furthermore it needed the right execution role in order to be allowed to publish to the AWS IoT message broker. In the end we received the message in our local broker and can conclude that it was successful.

lambda_function

```
1  import boto3
2  import json
3
4  def lambda_handler(event, context):
5
6      client = boto3.client('iot-data')
7
8      # Change topic, qos and payload
9      response = client.publish(
10         topic='commands/mir01',
11         qos=1,
12         payload=json.dumps({"COMMAND":"MOVE_DESK"})
13     )
```

Figure 15: Publishing an MQTT message with AWS Lambda over HTTP.

6.4 System Integration

Using the developed MQTT broker and Z-Wave API we could begin the process of combining all the parts into a single unified system. The main challenge was to complete the pipeline from receiving data from the sensors to controlling the MiR100.

6.4.1 Sending Sensor Data

Before being able to control the AGV, the data from the sensors had to be sorted and sent to the AWS cloud. This was done by iterating over every sensor registered in the Z-Wave network and creating a list of Client objects. The client objects can then be assigned different configurations and linked to separate topics.

Listing 6.7: Client object assignment

```
zw = ZWave.ZWave(config='../api_zwave/config')
zw.start_network()

sensors = zw.get_every_sensor_data()

for sensor_name in sensors.keys():
    client = Client(str(sensor_name))
    clients.append(client)

for client in clients:
    client.create()
```

Description of the code in Listing 6.7:

1. A Z-Wave network object is created and the network is started.
2. Using the created API function *get_every_sensor_data()*, all the sensors are gathered into a Python list object.
3. Looping on the key values in the list of sensors, a new client object is created and appended to the list of clients.
4. The client list is iterated and the *client.create()* function is called. This function takes care of all the paho and MQTT configuration that is required.

After all the clients has been created and configured, a loop is initiated to poll the sensors for data every 5 seconds, see Listing 6.8. The value is set to 5 seconds because this was the shortest supported polling interval for these specific nodes. If desired, the sleep function can be lowered, but no new sensor data would be gained as it would only collect the cached values.

Listing 6.8: Fetch sensor values

```
while True:
    data = zw.get_every_sensor_data() # as json
    for key,value in data.items():
        if name_contains(str(key), '6'):
            clients[0].publish(topic = 'sensor/aeotec06',
                               payload = json.dumps(value))
        elif name_contains(str(key), '8'):
            clients[1].publish(topic = 'sensor/aeotec08',
                               payload = json.dumps(value))
    sleep(5)
```

The sensors are assigned names based on the manufacturer and node id. This way we could easily differentiate the nodes based on their id, and use a simple if/else check to publish their values to their respective topic.

The published data would go through the MQTT broker and reach the AWS cloud. The received data was then handled by IoT rules, Figure 16.

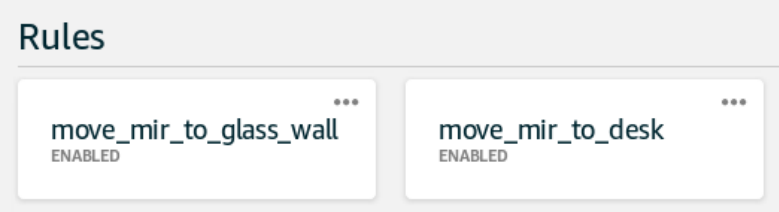


Figure 16: AWS IoT rules corresponding to their action

For each rule there exists an associated SQL statement which define when a rule action should trigger. These actions are able to connect with other AWS services. In this case, an action was created to invoke an AWS Lambda function (Figure 15). This publishes a command that will be parsed by a subscriber that is configured to call the MiR100. For testing purposes, a statement was created to call the MiR100 when the aeotec06 sensor reported a luminance value above 300, as seen in Figure 17. Modifying the luminance value proved to be one of the fastest way to test statements as it is easy to control this parameter. There was also created a similar statement for the aeotec08 sensor, which was set to trigger when the reported luminance was below 50.

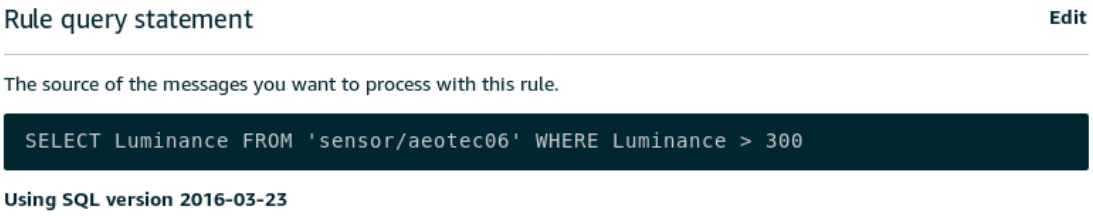


Figure 17: AWS rule query statement

6.4.2 Controlling the AGV

After successfully sending and receiving data from the sensors, the last required step was to control the AGV. The AGV is controlled by so called missions. These missions are what allows the unit to operate automatically and are represented with their own globally unique identifier (GUID). Missions has to be predefined by a user either via the API or using the graphical interface. Missions can include actions such as taxi services (move to a specific location), flashing a light or playing a sound. The MiR100 can be equipped with extra hardware such as an arm for picking up and placing objects. An example mission from the MiR100 mission interface reference guide can be seen in Figure 18.

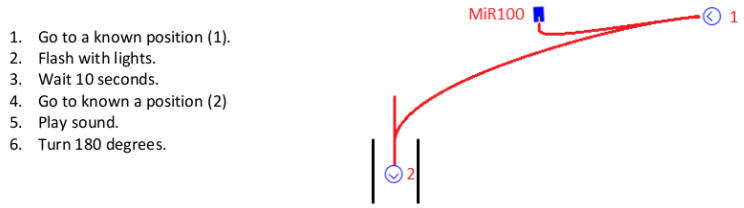


Figure 18: MiR100 example mission

Knowing that the AGV could be controlled by a REST API, we started testing some curl commands to get it moving. After some trial and error related to passing the correct payload, we were able to get the AGV moving by queuing a mission using the following command as seen in Listing 6.9.

Listing 6.9: curl command for activating a specific mission for the AGV

```
curl -H 'Content-Type': 'application/json' -X POST \
     -d '{'mission': '<unique_mission_id>'}' \
     http://<ip_addr>/v1.0.0/mission_queue \
```

In this project Escio had provided some preconfigured taxi missions. These missions are configured to drive between predefined locations, perfect for testing a prototype.

After receiving messages from the Paho broker and creating a MiR100 command processor, we designed our own MiR API. This adds the functionality of handling the HTTP requests to the MiR100 REST API. This can be integrated with the command processor adding a seperation of logic and allowing for modularity. The command processor functioned as a dictionary to look up the different

missions that existed at the given time. As there was only two missions we had access to, we created commands that corresponds to the messages received from the Lambda function, see Listing 6.10.

Listing 6.10: Python AGV commands

```
CMD_MOVE_TO_DESK = "MOVE_DESK"
CMD_MOVE_TO_GLASSWALL = "MOVE_WALL"
```

The command processing class is using Pahos callback function *on_message()* for handling incoming messages, shown in Listing 6.11. The received messages is of type MQTTMessage object and has to be decoded to UTF-8 before converted to JavaScript Object Notation (JSON).

Listing 6.11: on_message() function listening for incomming messages

```
def on_message(client, userdata, message):
    if message.topic == AGVCommandProcessor.commands_topic:
        try:
            message_dictionary =
                json.loads(message.payload.decode())
            ...
```

The if statement in Listing 6.12 checks to see if the message contains any valid commands. In the case of being a valid command an if statement is used with the MiR API to move the AGV to its requested location.

Listing 6.12: Checking the message for command keys

```
...

if COMMAND_KEY in message_dictionary:
    command = message_dictionary[COMMAND_KEY]
    agv = AGVCommandProcessor.active_instance.agv
    is_command_processed = False
    if command == CMD_MOVE_TO_DESK:
        agv.move_to_location("desk")
        is_command_processed = True
    elif command == CMD_MOVE_TO_GLASSWALL:
        agv.move_to_location("glasswall")
        is_command_processed = True
    ...
```

The *move_to_location()* method in Listing 6.13 is what invokes the MiR100 REST API and makes the unit move. It receives the commands from the AGVCommandProcessor and sends an HTTP request with the correct mission GUID to MiR100's system.

Listing 6.13: Method for calling the AGV in our MiR API

```
def move_to_location(self, mission, location=None):
    try:
        mission_queue = self.mir_url + '/mission_queue'
        data = {}
        if mission == 'desk':
            data = {'mission': \
                'fdffa984-4188-11e9-8674-f44d306ef756'}
        if mission == 'glasswall':
            data = {'mission': \
                'edd5cc25-4188-11e9-8674-f44d306ef756'}
        response = requests.post(mission_queue,
            headers=self.header, data=json.dumps(data))
        ...
```

In Figure 19, the command MOVE_WALL was received. The mission is then added to MiR100's internal queue and its status is updated to pending. The movement of the MiR100 can be seen in Figure 20. The blue square is the MiR100 and the yellow line is the front side of the unit. The blue circles represents preconfigured locations, and the green line is the current route it is taking. The gray and black areas are seen as part of the environment and the red lines are seen as dynamic, e.g. people and chairs.

```
Message received: {"COMMAND": "MOVE_WALL"}
{'actions': [],
 'control_posid': None,
 'control_state': 0,
 'finished': None,
 'id': 315,
 'max_action_id': 0,
 'message': '',
 'mission': '/v1.0.0/missions/edd5cc25-4188-11e9-8674-f44d306ef756',
 'mission_id': 'edd5cc25-4188-11e9-8674-f44d306ef756',
 'ordered': '2019-05-15T11:04:34',
 'parameters': '/v1.0.0/mission_queue/315/parameters',
 'started': None,
 'state': 'Pending'}
Moving to glasswall
```

Figure 19: Move to glasswall command received from AWS lambda

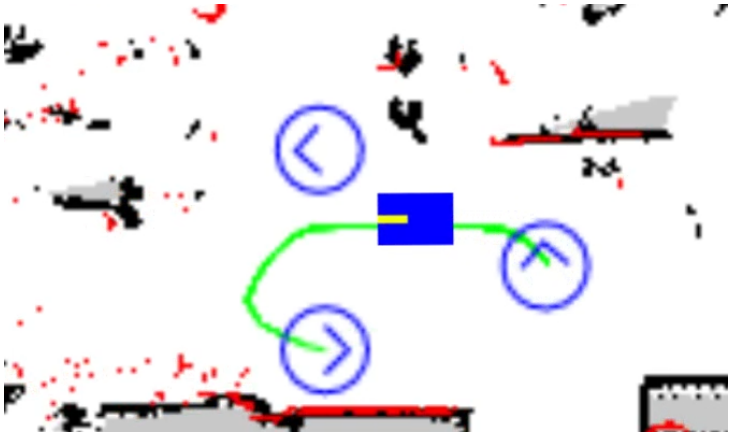


Figure 20: MiR100 moving to glasswall location

7 Development process

Following is an explanation of how the group worked during the implementation process. The chapter adresses the Scrum development modell and various tools for simplifying the process, code documentation and testing. We also explain the live product demo that we held in front of Escio employees.

7.1 Development Tools

Due to the requirements set by Escio in Section 3.3, an agile development process was selected. Working agile allows a flexible development plan, and as requirements and specifications changed, we were able to quickly react and adjust the priorities.

7.1.1 Scrum

Scrum is an agile methodology where products are built in a series of fixed-length iterations. There are four pillars that bring structure to this framework:

1. Sprint Planning
2. Daily Scrums
3. Sprints
4. Sprint Retrospective

During the start of the project we were doing sprints with a duration of 3 weeks (15 working days). This was partially because of the time constraints of the product owner and new technologies that required a substantially amount of research. This was later changed to a 2 + 1 week sprint duration, were 2 weeks were set aside for working on the project, and 1 week were set aside for research.

Before the start of each sprint we held sprint planning meetings to outline what needed to be done during the coming weeks. Tasks that had not been completed during the previous sprint that were still required took priority. If, during meetings or sprint retrospectives, requirements had changed or new functionality were requested, we had to weigh these against already existing tasks and possibly change the priority.

After planning and setting up the sprint backlog we played planning poker. This process proved to be very useful to us for a couple of reasons: It gave us a way to estimate task complexity, provided a way to manage time for each task, and let us discover tasks that were too big or ambiguous. We applied planning poker like this:

1. A user story was selected from the sprint backlog.
2. Each member had "cards" with different score values. These were used to represent the complexity and time requirement to complete the described task. The scores were assigned in a variant of Fibonacci, where the lowest score was 0, and the highest was 101.

3. The selected user story was discussed in detail and each member selected a card to represent their estimate.
4. If all the members gave the same score, this became the estimate of the user story. If members had different scores, we discussed the estimates between ourselves. If there was a large span between the scores, the task was discussed in more detail and alternatively altered.
5. This process was repeated until all the user story in the sprint backlog had been covered.

During each sprint we had daily standups to update each other on what work has been done and what work remains. This helped us with keeping track of different problems and issues that occurred. The daily standups were either held physically when working together, or via Discord when working remotely.

After each sprint we had a sprint meeting with Terje. These meetings were mostly meant to show the sprint results and talk about the plans for the next sprint. These meetings were also where most of the new requirements would surface. After the sprint meeting we usually set aside time to work on the next sprint backlog and materialize tasks into user stories.

7.1.2 Jira Software

Jira Software is an agile project management tool that supports agile methodologies. With this tool we were able to plan, track, and manage the entirety of our development process.

Jira is managed by creating a new project in the management dashboard. This initiates an empty project with an associated backlog. The team then starts filling up the backlog with the tasks needed. The first week Terje helped us formulating tasks into user stories, as this turned out to be more complicated than initially assessed. User stories are structured using the following formula:

> Persona: e.g. *As a <someone/role>.*.
> Need: e.g. *...I would like <do something>...*
> Purpose: e.g. *...so that I can <reach a goal/get some benefit>.*

An example of a user story for this project would be, for example: "*As a user I would like to gather data from Z-Wave devices so that it can be published to AWS*".

As mentioned earlier, some of the user stories was a challenge to articulate. The stories had to be easy to understand, descriptive and a way to abstractify the technical aspects. To help with this, Jira enables you to create Tasks that are more aimed towards the developers, however, mixing tasks and user stories should be done with care. What we did for some of the stories was breaking it down to sub-tasks and appending to the story. This way, we could have both user stories and tasks.

For the planning poker process, Jira has a built in tool called Agile Poker. This was, as mentioned, used before starting a sprint to ensure good planning and to discover potentially challenging tasks. This also had to be done in order to display the burndown chart correctly at the end of each sprint.

Jira Software has support for integration with different version control systems such as Bitbucket. This meant we were able to commit changes directly to different issues/tasks linked to the current running sprint backlog. This allowed for easier tracking and cooperation as each member was able to see what the other members were currently working on and their progress.

7.1.3 Trello

Trello[1] is a collaboration tool that organizes projects into boards. And while we used Jira for this, Trello was used as a support tool to gather all TODOS to help us visualize the sprint scope. These tasks where later converted into user stories and added to our Jira board. We also used it to keep track of minor sub-tasks that different functions needed in order to work correctly.

7.1.4 Bitbucket

Bitbucket[2] is a Git repository management solution. It gives us the ability to collaborate on our source code and handles development flow. It provides some commonly used features such as: access and workflow control, pull requests and Jira integration.

We used Bitbucket throughout the entirety of the development process. New features, bug fixes and added functionality were handled by creating pull requests. This allowed the other members to review the code and suggest changes before integrating it with the main branch. Each pull request had to be approved by at least one other team member.

7.2 Testing

During the development process we wrote some unit-tests for the REST API and performed user testing for the Z-Wave API. The unit-tests were limited because of the difficulties of testing network protocols and REST services. We also had a live product demonstration for Escio who was really happy with the results.

7.2.1 Unit-testing

Writing unit-tests for REST API was rather complicated. We decided to write tests for some of the less complex operations where it was feasible to generate mock data that would resemble the original payload. Many of the functions also returns different status codes, which has to be tested. The project supervisor said that testing is important, but if it was at the expense of the end-product, having a reduced test coverage is acceptable.

No unit-tests were written for the Z-Wave API as it meant setting up a dummy network which would not be feasible. Testing the MQTT protocol involves the verification of 4 central concepts:

1. Handling connection failure
2. Successful connection with and without the broker

[1] https://trello.com
[2] https://bitbucket.org

3. Correct parsing of the MQTTMessage object
4. Exception handling of malformed payloads

One does not usually want to test other services, as they are supposed to be responsible for their own testing. With this in mind, we decided against testing of the MQTT protocol for this project as it would require an excessive amount of time to set up a working test environment.

7.2.2 User testing

At the end of our last sprint we wiped clean the Raspberry Pi and unpaired all the Z-Wave devices. This was done in order to examine if the Bitbucket documentation (Section 7.3.3) was instructional enough for a user without prior IoT knowledge to understand and follow. The user testing provided us with some meaningful feedback. Some of the feedback we got was that the documentation was hard to follow at times and had an occasional lack of detailed description. Based on the feedback we have gotten, we have described suggested paragraphs with greater details.

7.3 Documentation

Documentation was very important both for us and the company. Having good documentation makes it easier for the end-user to get things up and running as intended. Every independent module was thoroughly documented using documentation tools such as Swagger and Sphinx.

7.3.1 Swagger

Swagger has a build in tool for providing user friendly documentation. Following the PEP-8 commenting standard, all documented code is parsed and rendered beautifully on the web page. This way, it makes the web page simpler to use for those who are not familiar with REST architecture.

7.3.2 Sphinx

Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl and licensed under the BSD license. It was originally created for the Python language so the support is outstanding and provides a ton of features such as HTML, PDF and LaTeXgeneration using makefiles. It has built-in support for code highlighting, cross referencing and more.

Sphinx was easily installed using a package manager, and provides a script that guides the user through the configuration process. The script creates an index.rst file which serves as the main configuration file that the user has to update to add documentation for different modules such as the Z-Wave API.

7.3.3 Bitbucket README

As a part of the documentation, we have made an in-depth guide on how to use our Z-Wave API. This guide provides a detailed description on how to configure the network and receive data from the sensors. Additionally we have provided screenshots of the expected outcome of executed commands. We have also described common mistakes and pitfalls that a user may encounter and how to avoid them. The README can be found in *Appendix C*.

7.4 Product Demo

After the completion of the last sprint, Terje invited us to present our prototype in front of the company. The demonstration consisted of the whole pipeline from getting the sensor data, to moving the AGV. The presentation proved to be very successful and Escio was happy with the end product.

8 Discussion

This chapter provides a summary of our results and reflect on various project aspects such as approach, management, project progress and teamwork.

8.1 Results

At the end of the project, we had developed three API's and produced desired results for the company. We were able to deliver both a REST API and an application for handling real-time sensor data to control the MIR100. Escio was very satisfied with our results.

8.1.1 REST API

We were able to complete the initial requirements for the REST API as well as providing additional functionality for future expansion. The user is able to use the API for sending messages and performing IoT core operations as shown in Section 6.1.

There were plans for adding support for Oath2 framework. This would enable us to limit the user access but had do be dropped due to time constraints. We also had some issues with support for additional parameters in the URLs, so an unconventional solution using the equals sign was provided. Additionally the Swagger UI is not configured to connect to the port used by Nginx, failing to reach the Django application.

8.1.2 AWS Communication

Communication between the AWS services and our system over the MQTT protocol using the Mosquitto broker was successful. The AWS Lambda triggers on preconfigured thresholds and notify our system to move the AGV. We created two solutions as shown in Figure 7 and Figure 14.

Although the Mosquitto broker proved to be successful, some security measures has to be taken as we did not have the time to implement this. The connection between the AWS IoT platform and our server is secured over TLS while the local connection is not. The server has security at the application level by requiring a username and password for accepting the request to connect. However, it is using a raw TCP/IP connection without security at the transportation level, and as a consequence the password and username is sent in plain text [60][61]. Mosquitto can be configured to only accept incoming request over TLS, however as mentioned we did not have time to implement this.

8.1.3 Z-Wave API

Overcoming a great deal of problems and challenges we managed to successfully implement the Z-Wave API. The API combines IoT technology and cloud computing for control and communication of an AGV unit. The challenges we faces and how we overcome them can be read further upon in

Section 6.2.

During the implementation phase we discovered that the procedure of configuring a device differs for wired and battery devices. We discovered that in order to configure a battery device it needed to be in an "awake" state throughout the configuration process. This however does not work for battery devices. As the battery devices have stricter requirements for battery preservation, they spend most of the time in the "sleeping" state. We discovered that their wake up interval is too short to be able to complete the configuration process.

8.2 Alternate choices

As described in Section 4.3 we chose Django as our web framework in favor of its clean and structured design. What we did not know back then was that Django's structure design comes with a cost of overhead. There needs to be done some preparation steps before the project can be used. Even after the setup step is finished there is this constant swap between different files and folders during the development. That comes in a natural way with a bigger project but, in our case it was more overhead than the actual work. So use of Flask might have been better choice in our case, based on the size of the project.

In retrospect, our choice of IoT protocol could have been different. We could have chosen any of the two other IoT protocols we mentioned in Section 4.6, mainly because of how bad the documentation for python-Openzwave is. Thread and ZigBee have great documentation[62][63] and the implementation process could have been much smoother.

8.3 Project Organization

The team had set up some responsibilities as described in Section 1.4.1. Here we will reflect on the overall project management, team dynamic and task distribution.

8.3.1 Group Organization

Workload varied a lot from week to week, and some weeks had more hours than others. We set up a secure shell (SSH) connection on the Raspberry Pi to be able to work from home. We created a small test setup using an Arduino and an led so that the luminance sensor could report dynamic data which could be accessed remotely. Team communication when working remotely was handled mostly via Discord using messages and voice chat. This way of working turned out fine as we all had worked on group projects before and knew each other very well. We also had previous experience with agile software approaches, so task designation was no issue.

Later on in the project when we started to work with the MiR100, we were invited to stay at Escio's location. They provided on-site seating with all the needed connections. They could also provide a monitor and peripherals for the Raspberry Pi. We were free to use this offer as often as we liked.

We had meetings on Thursdays every third week with Terje. Since Terje was on paternity leave during

the project, so additional meetings had to be planned in advance. Communication was managed mostly through mail, but also sometimes through mobile and text messages. Meetings with Hao were held when we felt the need. These meetings were planned via mail and held on the school or via Skype.

For the project we also required access to different AWS services. The access is handled via AWS IAM and had to be done by an official employee at Escio. As mentioned in Section 1.4.1, Anders were assigned the task to communicate what we needed with this person. This was difficult at times because we had to research what type of access we needed in advance and construct a mail containing all the access types we needed. Many of the services are closely linked together, and using one service might require access to another service. An example of this was when we requested access to all the IoT core services, but was not able to display the data flow because we did not have sufficient access to the Amazon Cloud Watch service.

Hao is not a native speaker of Norwegian so the decision to write the report in English was made by the group. This way, Hao could help us with some of the spelling corrections and sentence phrasing. Naturally, all the meetings was also held in English.

Overleaf was utilized to write the report as this provides real-time collaboration, chat services and a history functionality. The report is written in LaTeX as this is what the group members are most familiar with.

The last sprint review was set to be the 11th of April, but as we were asked to have a product presentation the 27th of April, we decided to set a hard deadline for finalizing the product before this date. This gave us just under one month to write the report.

8.3.2 Evaluation of Jira

As described in Section 7.1.2, Jira uses tasks and user stories. The problem we had with Jira was that it was difficult at times to split user stories into smaller chunks. It was possible to assign multiple tasks to a specific user story but those tasks could not be assigned any planning poker points. This led to big jumps on the burnout chart as only the finalizing of a user story updated the burnout chart. The effect of user storied on burnout charts can be seen in *Appendix D*

8.3.3 Assignment of Project Tasks

Working with Jira means that group members selects the tasks they want to work with the most, prioritizing blocking features. When developing the REST API, each member chose what they wanted to implement. When the scope got extended we had to decide on who should do what in order to make sure the project would be completed. The major assignation happened when working with MQTT and Z-Wave. Two of the group members where assigned to implement the Z-Wave API, while the other member was tasked to implement the communication between our system and AWS IoT.

9 Conclusion

Including the use of IoT technology in robotics has the benefit of automating processes that are currently being done manually. We have shown through our developed prototype that the combination of these two technologies is possible and provides a feasible solution that can help reduce manual labour.

The goal of this project, described in Section 1.2, was to create a prototype of a unified system for IoT sensor data transmission and controlling an AGV based on this data. By utilizing currently available IoT and robotic technologies, we have implemented a system that allows for requesting an AGV to predefined locations. Our solution makes use of the Z-Wave IoT protocol for sensor reporting and publishes these values over the MQTT protocol to AWS. This data separated into topics and handled by IoT rules which triggers an AWS Lambda function to control the AGV. The conclusion is that it is achievable to control an AGV unit based on sensor data from an IoT network. The company is very satisfied with our research of the topics, implementation process and produced results.

9.1 Future Work

To further improve on the prototype, several additions can be made. One of additions would be to automatize the process of detection and pairing of new devices. The Raspberry Pi along with the Z-Stick could be mounted on top of the AGV. The AGV would further be programmed to drive and scan the environment around it. The Z-Stick would automatically pair with the devices it detects and add them to the MiR's location map. This feature would be useful in areas like big warehouses where dozens of sensors are already installed and they all needs to be paired.

Another improvement that could be done would be to ensure security regarding all parts of the project. The Z-Wave protocol we were using in the development process has the support for encryption, but it was not our focus for the prototype phase. Enabling and actively utilizing the supported encryption on the Z-Wave network, as well as establishing a secure connection with other elements in the pipeline, would be a great way to provide security.

As of now, all interaction with the Z-Wave API is made via the command line interface. A graphical user interface would be a way to increase the overall user experience and ease of use. Additionally, it would serve as a way to provide a cleaner overview for various connected devices, statistics and analysis, as well as live monitoring.

Bibliography

[1] Lara De Schutter. 9 things you need to know about mesh networks. https://hackernoon.com/9-things-you-need-to-know-about-mesh-networks-f61a77e5751a, 2018. [Online; last accessed 18-May-2019].

[2] Shahin Farahani. *ZigBee Wireless Networks and Transceivers*. Newnes, 2008.

[3] H. A. A. Al-Kashoash and Andrew H. Kemp. Comparison of 6lowpan and lpwan for the internet of things. *Australian Journal of Electrical and Electronics Engineering*, 13(4):268–274, 2016.

[4] Asad Munir. Z-wave technology. https://spectrum.co.ae/blogs/technology/z-wave-technology, 2017. [Online; last accessed 18-May-2019].

[5] MiR. MiR robot REST API. https://www.mobile-industrial-robots.com/media/1411/mir-rest-api-reference-guide-100.pdf, 2018. [Online; last accessed 18-May-2019].

[6] Biranavan Pulendralingam, David Cerny, Emil Blixt Hansen, Rasmus Andersen, and Steffen Madsen. Autonomous industrial mobile manipulator in smart production. https://blixtdevelopment.com/pdf/Robotics_P5.pdf, January 2017. [Online; last accessed 18-May-2019].

[7] Wikipedia contributors. Internet of things — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Internet_of_things&oldid=894621947, 2019. [Online; last accessed 18-May-2019].

[8] Rob Barton, David Hanes, and Gonzalo Salgueiro. *IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things*. Cisco Press, 2017.

[9] ISO/IEC stage 20922:201. Information technology – message queuing telemetry transport (mqtt) v3.1.1. https://www.iso.org/standard/69466.html, 2016. [Online; last accessed 18-May-2019].

[10] Gastón C. Hillar. *MQTT Essentials - A Lightweight IoT Protocol*. Packt Publishing, 2017.

[11] Bryan Boyd, Joel Gauci, Vasfi Gucer, Rahul Gupta, Vladimir Kislicins, Michael P Robertson, and Nguyen Van Duy. *Building Real-time Mobile Solutions with MQTT and IBM MessageSight*. IBM Redbooks, 2014.

[12] BISCI. *Network Design Basics for Cabling Professionals*. McGraw-Hill Professional, 2002.

[13] Wikipedia contributors. Network topology — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Network_topology&oldid=896144777, 2019. [Online; last accessed 18-May-2019].

[14] Wikipedia contributors. Z-wave — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Z-Wave&oldid=886885917, 2019. [Online; last accessed 18-May-2019].

[15] Wikipedia contributors. Mesh networking — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Mesh_networking&oldid=887391196, 2019. [Online; last accessed 18-May-2019].

[16] Torry Harris. Cloud computing–an overview. https://www.thbs.com/downloads/Cloud-Computing-Overview.pdf, December 2013. [Online; last accessed 18-May-2019].

[17] *Cloud Computing Guidelines*. Ministry of Transport and Communications, September 2017.

[18] Ministry of Local Government and Modernisation. Cloud Computing Strategy for Norway. https://www.regjeringen.no/contentassets/4e30afec51734d458596e723c0bdea0e/cloud_computing_strategy.pdf, 2016. [Online; last accessed 19-May-2019].

[19] What is cloud computing? https://aws.amazon.com/what-is-cloud-computing/, May 2019. [Online; last accessed 18-May-2019].

[20] Brad A. Myers and Jeffrey Stylos. Improving api usability. *School of Computer Science, Carnegie Mellon University*, 2015.

[21] Walid Maalej and Martin P Robillard. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.

[22] Bill Kleyman. Understanding cloud apis, and why they matter. https://www.datacenterknowledge.com/archives/2012/10/16/understanding-cloud-integration-a-look-at-apis, October 2012. [Online; last accessed 18-May-2019].

[23] Wikipedia contributors. Go (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Go_(programming_language)&oldid=895187560, 2019. [Online; last accessed 18-May-2019].

[24] Google and contributors. The go programming language. https://golang.org/doc/faq#goroutines, 2019. [Online; last accessed 18-May-2019].

[25] Vlad Guinard, Dominique; Trifa. *Building the Web of Things: With examples in Node.js and Raspberry Pi*. Manning Publications Co., 2016.

[26] Charles R. Severance. *Python for Everybody: Exploring Data Using Python 3*. Independent, 2016.

[27] Python Software Foundation. Web frameworks for python. https://wiki.python.org/moin/WebFrameworks, 2019. [Online; last accessed 18-May-2019].

[28] Mirjana Maksimović, Vladimir Vujović, Nikola Davidović, Vladimir Milošević, and Branko Perišić. Raspberry pi as internet of things hardware: performances and constraints. *design issues*, 3(8), 2014.

[29] Raj Ven. Why java is the perfect choice for iot. https://oneteam.us/why-java-is-the-perfect-choice-for-iot/, 2018. [Online; last accessed 18-May-2019].

[30] Aleksei Voitylov. The status of java on arm. *BellSoft*, January 2019.

[31] Mark Masse. *REST API Design Rulebook*. O'Reilly Media, 2011.

[32] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

[33] Sebastian Eschweiler. Rest vs. graphql. https://medium.com/codingthesmartway-com-blog/rest-vs-graphql-418eac2e3083, 2018. [Online; last accessed 18-May-2019].

[34] David Lord. Flask. https://github.com/pallets/flask, 2018. [Online; last accessed 18-May-2019].

[35] Daniel Rubio. Rest services with django. In *Beginning Django*, pages 549–566. Springer, 2017.

[36] Justin Ellingwood. Apache vs nginx: Practical considerations. https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations, January 2015. [Online; last accessed 18-May-2019].

[37] Alexandra Leslie. Nginx vs. apache (pro/con review, uses, & hosting for each). https://www.hostingadvice.com/how-to/nginx-vs-apache/, January 2018. [Online; last accessed 18-May-2019].

[38] Robin Muilwijk. Top 5 open source web servers. https://opensource.com/business/16/8/top-5-open-source-web-servers, 2016. [Online; last accessed 18-May-2019].

[39] Wikipedia contributors. Zigbee — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Zigbee&oldid=891079669, 2019. [Online; last accessed 18-May-2019].

[40] ZigBee Alliance. Zigbee specification. Zigbee document 053474r20, ZigBee Standards Organization, September 2012. https://www.zigbee.org/wp-content/uploads/2014/11/docs-05-3474-20-0csg-zigbee-specification.pdf.

[41] ZigBee Alliance. Zigbee specification faq. https://web.archive.org/web/20130627172453/http://www.zigbee.org/Specifications/ZigBee/FAQ.aspx, 2013. [Online; last accessed 18-May-2019].

[42] NTS. Top 5 reasons for failures in zigbee deployments. https://www.nts.com/resources/Top%205%20Reasons%20for%20Failures%20in%20ZigBee%20Deployments.pdf, 2012. [Online; last accessed 18-May-2019].

[43] Gaotao Shi and Keqiu Li. *Signal Interference in WiFi and ZigBee Networks*. Springer International Publishing, 2017.

[44] Wikipedia contributors. Thread (network protocol) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Thread_(network_protocol)&oldid=894786733, 2019. [Online; last accessed 18-May-2019].

[45] Wikipedia contributors. 6lowpan — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=6LoWPAN&oldid=895359412, 2019. [Online; last accessed 18-May-2019].

[46] Z-Wave Alliance. About z-wave technology. https://z-wavealliance.org/about_z-wave_technology, 2019. [Online; last accessed 18-May-2019].

[47] Eric Brown. Z-wave opens up with new public sdk and developer site. http://linuxgizmos.com/z-wave-opens-up-with-new-public-sdk-and-developer-site, 2018. [Online; last accessed 18-May-2019].

[48] python openzwave. Python wrapper for openzwave. https://github.com/OpenZWave/python-openzwave, 2019. [Online; last accessed 18-May-2019].

[49] nthyagarajan. Python sample for z-ware web api. https://github.com/Z-WavePublic/PyZWare, 2018. [Online; last accessed 18-May-2019].

[50] king dopey. Pytomation is an extensible device communication and automation system written in python. https://github.com/king-dopey/pytomation, 2018. [Online; last accessed 18-May-2019].

[51] Ian Sommerville. *Software Engineering*. Pearson Education, 2016.

[52] Yves Chevallier. How to read/write configuration on a z-wave device? https://github.com/OpenZWave/python-openzwave/issues/121, 2018. [Online; last accessed 18-May-2019].

[53] David Vassallo. Z-wave : Lessons learned – python openzwave. http://blog.davidvassallo.me/2018/12/06/z-wave-lessons-learned-python-openzwave, 2018. [Online; last accessed 18-May-2019].

[54] Jonathan Kozolchyk. How to prepare for aws's move to its own certificate authority. https://aws.amazon.com/blogs/security/how-to-prepare-for-aws-move-to-its-own-certificate-authority/, March 2018. [Online; last accessed 18-May-2019].

[55] Mqtt frequently asked questions. https://mqtt.org/faq, May 2019. [Online; last accessed 18-May-2019].

[56] Service name and transport protocol port number registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=8883, May 2019. [Online; last accessed 18-May-2019].

[57] Michael Garcia. How to bridge mosquitto mqtt broker to aws iot. https://aws.amazon.com/blogs/iot/how-to-bridge-mosquitto-mqtt-broker-to-aws-iot/, August 2016. [Online; last accessed 18-May-2019].

[58] Aws lambda features. https://aws.amazon.com/lambda/features/, 2019. [Online; last accessed 18-May-2019].

[59] Lambda function to connect to aws iot and publish mqtt messages. https://github.com/aws/aws-iot-device-sdk-js/issues/97, November 2016. [Online; last accessed 18-May-2019].

[60] The HiveMQ Team. Mqtt essentials part 3: Client, broker and connection establishment. https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/, January 2015. [Online; last accessed 19-May-2019].

[61] The HiveMQ Team. Introducing the mqtt security fundamentals. https://www.hivemq.com/blog/introducing-the-mqtt-security-fundamentals/, April 2015. [Online; last accessed 19-May-2019].

[62] Nest Labs. Get started. https://openthread.io/guides, 2019. [Online; last accessed 18-May-2019].

[63] Digi International Inc. Get started with xbee python library. https://xbplib.readthedocs.io/en/latest/getting_started_with_xbee_python_library.html, 2018. [Online; last accessed 18-May-2019].

Appendices

A Project Description - Original

escio

Vi tar ansvaret!

API for IoT-data

Web-basert API for streaming av data til AWS IoT

1 Oppdragsgiver og kontaktinformasjon

Oppdragsgiver

Escio AS
Nedre Torvgate 6 (NT6)
2816 Gjøvik

post@escio.no
Tlf: 47 47 25 80

Kontaktperson

Terje Krogstad, prosjektleder
terje@escio.no
Tlf: 959 72 382

2 Beskrivelse

Når man skal prototype IoT-applikasjoner som skal sende sensordata til skytjenester som AWS (Amazon Web Services) bruker man ofte mikrokontrollere eller SoC (System On Chip) som kommuniserer med AWS via AWS IoT over MQTT.

I dette prosjektet ønsker vi å generalisere et API som i sanntid kan ta imot data og strømme dette videre til AWS via AWS IoT. APIet skal være HTTP-basert og basere seg på industristandarder (REST/Swagger eller GraphQL) slik at det enkelt kan integreres mot systemer og tjenester som har behov for å sende data inn. Det kan være behov for å sette opp tjenester for å mellomlagre dataene som strømmes gjennom APIet, men dette kan vi sammen med prosjektgruppa se om er mulig å gjennomføre innenfor prosjektperioden.

All validering og ruting av data videre til lagringstjenester i AWS skal håndteres med regler i AWS IoT, noe som gjør at det ikke er behov for å ta høyde for dette i prosjektet.

3 Oppgaver / Mål

- Research i forhold til tekniske løsninger
- Utvikle en prototype på et API som tar imot data (I første omgang to dimensjonale data som f.eks. tid og verdi) og streamer dette videre til AWS IoT
- Valgfritt: Sette opp en caching-tjeneste som applikasjonen med APIet kan logge hendelser og strømmedata mot

B Project Description - Extended

escio

Vi tar ansvaret!

Sensorstyrt bestilling av mobilitetsenhet

AWS IoT-basert system for bestilling av AGV til spesifikke lokasjoner

1 Oppdragsgiver og kontaktinformasjon

Oppdragsgiver

Escio AS
Nedre Torvgate 6 (NT6)
2816 Gjøvik

post@escio.no
Tlf: 47 47 25 80

Kontaktperson

Terje Krogstad, prosjektleder
terje@escio.no
Tlf: 959 72 382

2 Beskrivelse

Oppgaven er i all hovedsak todelt.

Fase 1:

Når man skal prototype IoT-applikasjoner som skal sende sensordata til skytjenester som AWS (Amazon Web Services) bruker man ofte mikrokontrollere eller SoC (System On Chip) som kommuniserer med AWS via AWS IoT over MQTT.

Vi ønsker å generalisere et API som i sanntid kan ta imot data og strømme dette videre til AWS via AWS IoT. APIet skal være HTTP-basert og basere seg på industristandarder (REST/Swagger eller GraphQL) slik at det enkelt kan integreres mot systemer og tjenester som har behov for å sende data inn. Det kan være behov for å sette opp tjenester for å mellomlagre dataene som strømmes gjennom APIet, men dette kan vi sammen med prosjektgruppa se om er mulig å gjennomføre innenfor prosjektperioden.

All validering og ruting av data videre til lagringstjenester i AWS skal håndteres med regler i AWS IoT, noe som gjør at det ikke er behov for å ta høyde for dette i prosjektet.

Fase 2:

Prototype et helhetlig system som muliggjør bestilling av AGV (Automated Guided Vehicle) mobilitetsenhet til spesifikke lokasjoner når data fra statiske sensorer passerer gitte prekonfigurerte terskler.

Dette innebærer å sette opp en arkitektur med sensorer i mesh-struktur som leverer data til AWS IoT. Det skal være mulig å konfigurere regler for å bestille en AGV til å kjøre til prekonfigurerte lokasjoner hvis data fra en gitt sensor passerer prekonfigurerte terskler. Det skal også være mulig å benytte test-APIet fra fase 1 for å simulere sensordata på lik linje med reelle sensordata.

Studentene vil få tilgang til AGVen MiR100 for å prototype løsningen.

3 Oppgaver / Mål

- Research i forhold til tekniske løsninger
- Utvikle en prototype på et API som tar imot data (I første omgang to dimensjonale data som f.eks. tid og verdi) og streamer dette videre til AWS IoT
- Valgfritt: Sette opp en caching-tjeneste som applikasjonen med APIet kan logge hendelser og strømmedata mot
- Utvikle en prototype på et system for bestilling av MiR100 til gitte lokasjoner basert på regler og terskelverdier for sensordata

C Z-Wave API Guide

Welcome

In this section we will guide you through the use of our Z-Wave API.

We have tested the Z-Wave API with the following setup:

- Raspberry Pi 2 & 3B
- Raspbian Stretch Lite+
- Python 3.5.3
- Aeotec Z-Stick Gen 5
- Aeotec MultiSensor 6
- Fibaro Motion Sensor

Our API is build upon the Python-openzwave wrapper for openzwave.

Pre-Setup

Before you can start using our Z-Wave API you need to do some preparation first. The first thing you need to do is to pair the Z-Wave devices with you Z-Stick. You can find the information on how to pair devices in you Z-Stick manual as well as you Z-Wave device manual. Example documentation(1)(2)(3)

Setup

After your Z-Wave devices are all paired with the Z-stick we can continue with further setup.

1. Power on your Z-Wave devices.
2. Connect your Z-Stick to the Raspberry Pi.
3. Check that the Z-Stick is detected by the Raspberry Pi. You can use the following command `ls /dev/`

The command should return bunch of different devices. In our case the Z-Stick device was displayed as **ttyACM0**. If you are having troubles with finding what your Z-Stick device is, simply disconnect the Z-Stick device from the Raspberry Pi, use the `ls /dev/` command then connect the Z-Stick on your Raspberry Pi and use the command again, that way you can see names of new devices popping up.

Z-Wave API

After the initial setup, we can finally make use of the API. Before you can use it, you just need to enter the **api_zwave** directory.

We recommend to start a python3 interpreter for the configuration process, it will be easier to follow along with the guide. The first thing you need to do is to import the zwave.py file and create an object of our Z-Wave class. Make sure you send your Z-Stick device path as a parameter like this:

```
from zwave import ZWave
zw = ZWave("/dev/ttyACM0")
```

The next step is to initialize the network. You can do it by calling the **start_network()** method.

Now that the network is started we can do all of different stuff with it. The first this you probably want to do is to get the data from sensors. Before you do that tho you might want to change the interval on how often your device should report the values. The default polling interval is set to somewhere around 1 hour, depends on the manufacturer of your Z-Wave device. Check your Z-Wave device manual for more information.

Firstly lets find out if the network is aware of all the connected devices. To do so call the **get_every_sensor_data()** method. This should return some information about your connected devices. Simply print the response, the output should look like this:


```
{
  "AEON_8": {
    "Sensor": true,
    "node_id": 8,
    "Temperature": 23.700000762939453,
    "Relative Humidity": 16.0,
    "Luminance": 24.0,
    "Ultraviolet": 0.0
  },
  "AEON_6": {
    "Sensor": true,
    "node_id": 6,
    "Temperature": 23.600000381469727,
    "Relative Humidity": 16.0,
    "Luminance": 23.0,
    "Ultraviolet": 0.0
  },
  "FIBARO_7": {
    "Sensor": true,
    "node_id": 7,
    "Seismic Intensity": 0.0,
    "Temperature": 28.200000762939453,
    "Luminance": 11.0
  }
}
```


You can see that the response returned some simple sensor values for each device. One that we are interested in right now is **node_id**.

Now we need to get the information about the device configuration in order to know which settings we can change on the device. Call the **print_configs()** method with the node_id for the device you wish to change, as a parameter.

In our example it was node 6 we wanted to change the configuration for. This resulted in the following output:


```
home_id: [0xcefb7bef] id: [72057594148815745] parent_id: [6] label: [Set the lower limit value of ultraviolet sensor] data: [4]
home_id: [0xcefb7bef] id: [72057594148816643] parent_id: [6] label: [Group 2 Interval] data: [3600]
home_id: [0xcefb7bef] id: [72057594148815492] parent_id: [6] label: [Report Only On Thresholds] data: [Disabled]
home_id: [0xcefb7bef] id: [72057594148818070] parent_id: [6] label: [Temperature Calibration] data: [1]
home_id: [0xcefb7bef] id: [72057594148814977] parent_id: [6] label: [Awake timeout] data: [30]
home_id: [0xcefb7bef] id: [72057594148818113] parent_id: [6] label: [Ultraviolet Calibration] data: [0]
home_id: [0xcefb7bef] id: [72057594148815507] parent_id: [6] label: [Temperature Reporting Threshold] data: [20]
home_id: [0xcefb7bef] id: [72057594148816148] parent_id: [6] label: [LED blinking report] data: [Enable LED blinking]
home_id: [0xcefb7bef] id: [72057594148815766] parent_id: [6] label: [Set the recover limit value of temperature sensor] data: [5121]
home_id: [0xcefb7bef] id: [72057594148815876] parent_id: [6] label: [Temperature scale] data: [Celsius]
home_id: [0xcefb7bef] id: [72057594148818081] parent_id: [6] label: [Humidity Calibration] data: [0]
home_id: [0xcefb7bef] id: [72057594148815651] parent_id: [6] label: [Set the lower limit value of temperature sensor] data: [1]
home_id: [0xcefb7bef] id: [72057594148814884] parent_id: [6] label: [Wake up 10 minutes on Power On] data: [Disable]
home_id: [0xcefb7bef] id: [72057594148815665] parent_id: [6] label: [Set the upper limit value of humidity sensor] data: [60]
home_id: [0xcefb7bef] id: [72057594148815793] parent_id: [6] label: [Set the recover limit value of Lighting sensor] data: [10]
home_id: [0xcefb7bef] id: [72057594148816499] parent_id: [6] label: [Group 3 Reports] data: [0]
home_id: [0xcefb7bef] id: [72057594148818102] parent_id: [6] label: [Luminance Calibration] data: [0]
home_id: [0xcefb7bef] id: [72057594148816627] parent_id: [6] label: [Group 1 Interval] data: [3600]
home_id: [0xcefb7bef] id: [72057594148815553] parent_id: [6] label: [Battery Reporting Threshold] data: [10]
home_id: [0xcefb7bef] id: [72057594148815617] parent_id: [6] label: [Enable/disable to send a report on Threshold] data: [0]
home_id: [0xcefb7bef] id: [72057594148815681] parent_id: [6] label: [Set the lower limit value of humidity sensor] data: [50]
```


You will get different configuration info for different devices. In this example we used Aeotec MultiSensor 6.

Let's say that we want to change the polling interval for the luminance. We can quickly discover that there is no "Luminance interval" among the listed values. Neither is there any "Temperature interval" or "Humidity interval". And this is where it gets interesting. If we look in the manual for the Aeotec Multisensor 6 we can see that there is no information about polling interval to the specific sensor type. On closer inspection we can see that there are some information about the "Report Table" and some "Groups". Those consists of different

integer values. In order to decode the values meaning we had to dive deep into the config files in the Config folder. In the Aeotec MultiSensor 6 config file we have found the meaning behind the integer values. It turns out that the values represents different sensor types. By default the value of Group 1 is set to 241 which means that it affects all of the sensor types. If you look closely on the result of print_configs() method, you can see that there are [Group 1 Interval], [Group 2 Interval] etc. what it means, is that if we want to change polling interval for the luminance we need to change the Group 1 polling interval and by doing this we will be changing the polling interval for all of the other sensors associated with.

The Fibaro Motion Sensor is way easier to configure. You can access luminance, temperature etc. directly without all the overhead associated with Aeotec sensor. It comes with another drawback tho. You can read more in the "Common Exception" section.

Now that we know all of this we can start to set some new polling intervals. We can do it by calling the **configure_node()** method which takes node_id and id as a parameter. In our case the method call will look like this **configure_node(6, 72057594148816627, 10).**

The method should return something like this:

```
Enabling poll on node 6
True
Polling enabled on node 6
Configuration completed
```

We need to stop the network in order to apply and save the new configuration.

After we turn the network on again we can double check if the config was successful by running **print_configs()** method.

```
home_id: [0xcefb7bef] id: [72057594148816627] parent_id: [6] label: [Group 1 Interval] data: [10]
```

We can see that the Group1 polling interval changed from 3600 to 10. Now that our polling interval is configured how we want it, we can start to get values from the devices. We can do it by calling the **get_sensor_data(6)** on the node we want to get values from or from every node available by calling the **get_every_sensor_data()** method.

After you are done, simply close the network by calling **network_stop()** method.

Common exceptions

Here are some exceptions that often happend to us.

1

If you are planning on connecting some specific devices on your Raspberry Pi in addition to Z-Stick, like arduino or wifi-adapter, be aware. The arduino in our case took the default **/dev/ttyACM0** spot of the Z-Stick, so when we wanted to start the network we always got the "Error establishing the network...". We advise you to check the Z-Stick device path after every new device you connect to your Raspberry Pi (except for the Z-Wave devices, they don't have any device path).

2

Don't take your Z-Stick device path for granted. If you have more devices connected to the Raspberry Pi like in the section above and you happen to reboot your Raspberry Pi, your Z-Stick device path might change. Check your Z-Stick device path after each reboot.

3

Battery devices are sleeping by default in order to save energy. In order to configure a device it needs to be awake. If you have a battery device you need to look up in you device manual on how to wake up your device. Your device should be awake at least 30 seconds in order to apply all configured changes.

D Burndown Charts

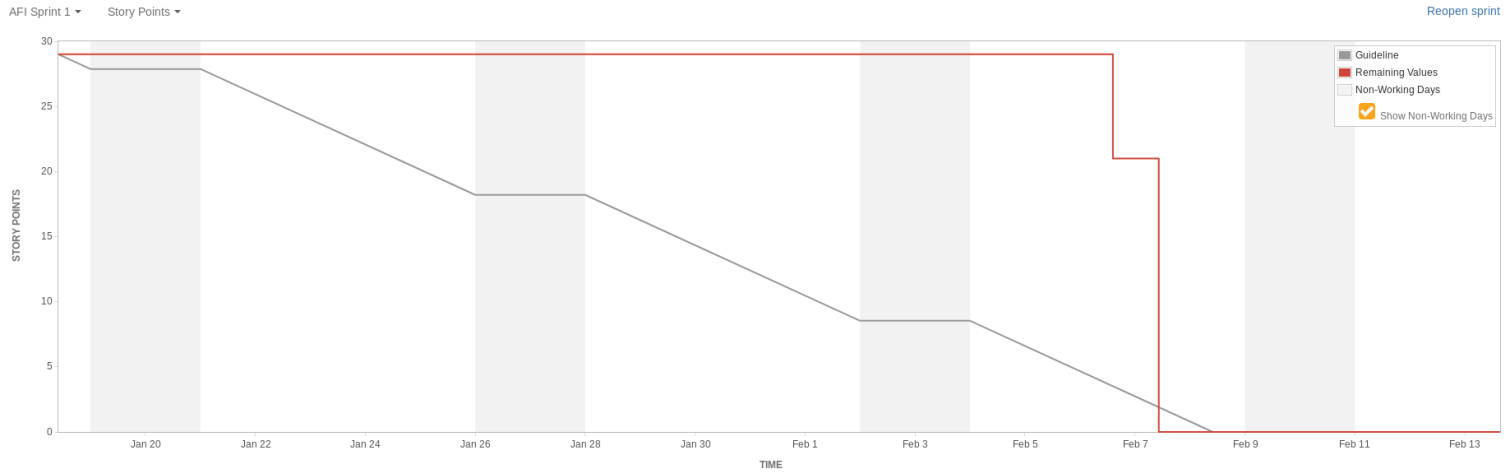


Figure 21: Burndown Chart for the first sprint

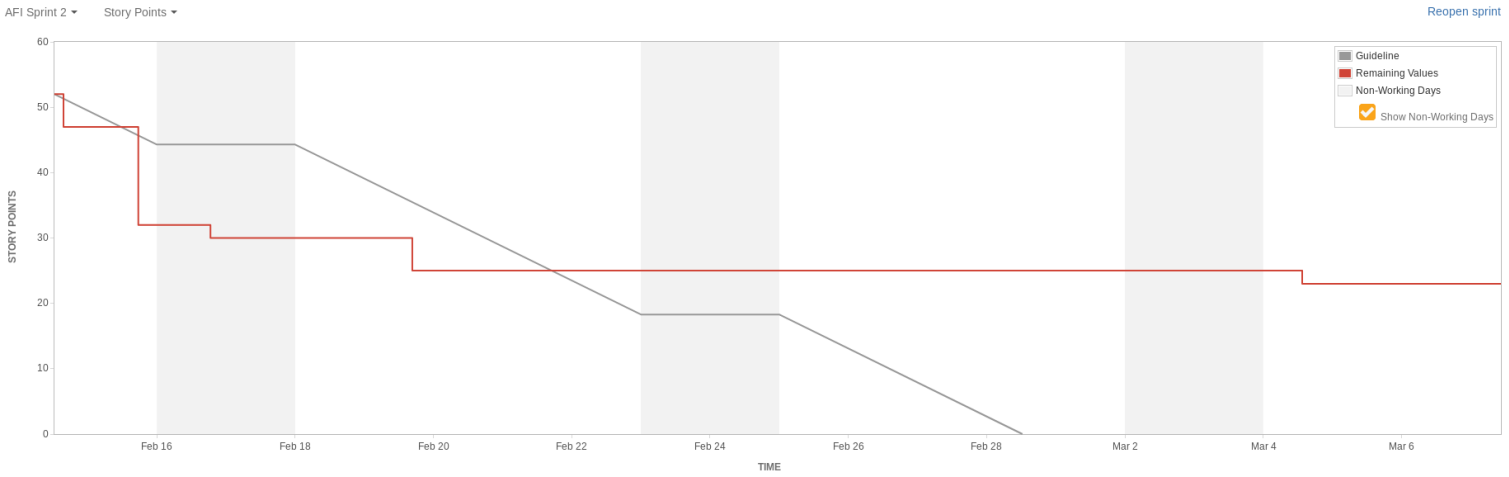


Figure 22: Burndown Chart for the second sprint

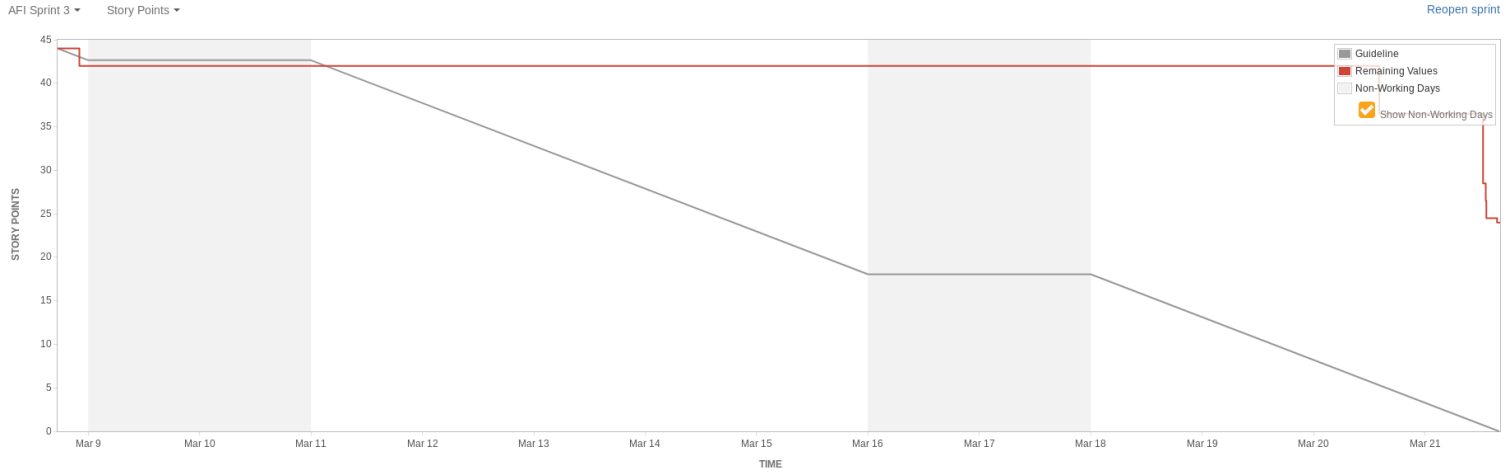


Figure 23: Burndown Chart for the third sprint

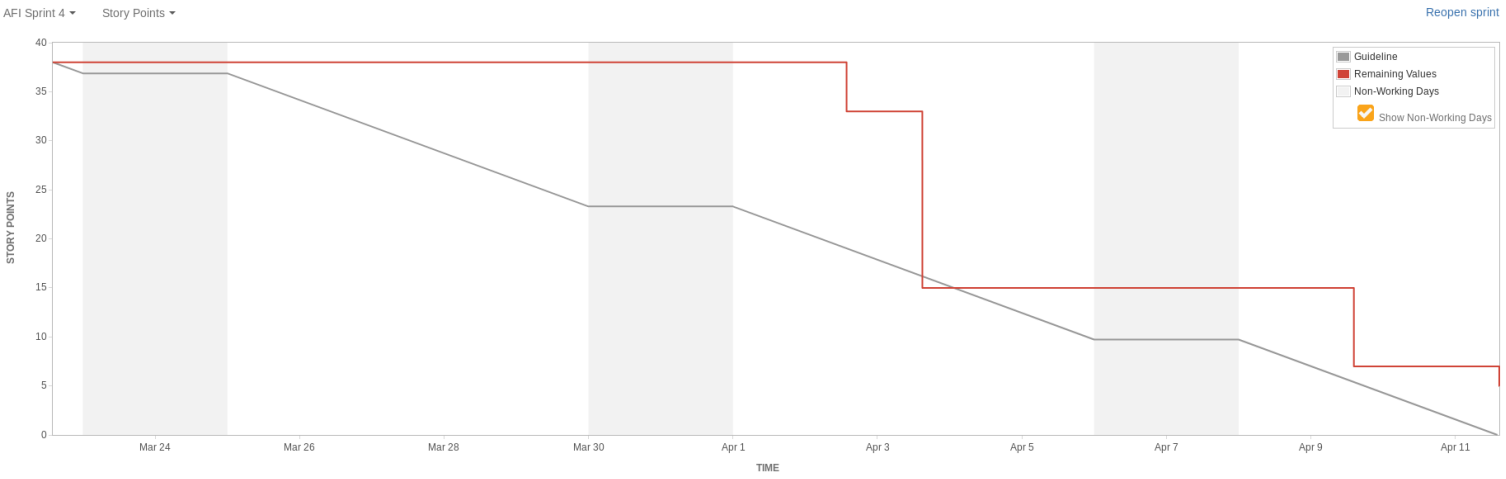


Figure 24: Burndown Chart for the fourth sprint

E Project plan

Project plan

Bartosz Tracz, Eirik Persson Masteholtet, Anders Isaksen

January 2019

Contents

1 Project Goals and Constraints

1.1 Background

In computer programming, an application programming interface (API) is a set of subroutine definitions, communication protocols, and tools for building software. In general terms, it is a set of clearly defined methods of communication among various components. A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer[1].

The Internet of Things (IoT) is the network of devices such as vehicles, and home appliances that contain electronics, software, actuators, and connectivity which allows these things to connect, interact and exchange data[2].

Amazon Web Services (AWS) is a comprehensive, evolving cloud computing platform. The idea behind cloud computing is simple: The user can simply use storage, computing power, or specially crafted development environments, without having to worry how these work internally. Cloud computing is usually Internet-based computing[3].

In this project we will mainly be using AWS' IoT Core service to develop the API.

1.2 Project Goals

The goal for this project is to develop a prototype for a functional REST API connected to AWS IoT Core service. The company wants the ability to connect and communicate with AWS using the existing HTTP-protocols. The API will provide two different services leading to two different main goals:

1.2.1 Prototype for the IoT-AWS Connection

- Allow users to perform CRUD actions on different part of the IoT framework
- Use a simple interface to connect and perform data operations
- Increase and decrease different users access rights
- Easily port the API to different systems using a container service such as Docker

1.2.2 Real-time Monitoring and Data Analysis

- Display the current dataflow
- Show data statistics
- Analyze the data sent over to AWS

After finishing the prototype the focus will be shifted towards data caching and expanding the APIs functionality. If time allows it we will begin researching the possibilities of edge computing to expand on the real-time aspects of the task.

1.3 Project Constraints

1.3.1 Development

For the prototyping of the application we have a lot of options. The languages we considered the most were Python and Go. Python mostly for ease of use, readability and good support with the Flask framework. All the members also have a lot of previous experience with Python, which makes it easy to get started on an application fast. Go is considered mostly because it is fast (runs straight on the hardware), has great support for multithreading and easy-to-read syntax. Python offers many tools to develop REST APIs, while Go has a strong standard library for web development.

We also considered C/C++, Java and NodeJS. C and C++ are omitted because Go is built on C and therefore inherits C's speed and efficiency.

Furthermore, C's syntax is more complex than Go and a lot less readable. Java is still being discussed as a potential language, but will most likely not be used in the prototyping phase. NodeJS was considered as it is based on Chrome's V8 which is fast and powerful, however, we would like to steer away from using JavaScript in this application due to its bad rep.

1.3.2 Time

Our last sprint meeting will be May 2nd where we will discuss the end product. Therefore we need to be finished with our prototype prior to this date. Secondly there is a deadline on the thesis on May 20th where the full project report needs to be delivered.

2 Scope

2.1 Project Description

The purpose of this project is to generalize an API which can receive data in real-time and further forward it to the AWS through AWS IoT. The API should rely on the HTTP-protocols and be based on industrial standards such as REST/Swagger or GraphQL.

Validation and routing of all data to the AWS must be handled with respect to the AWS IoT policies, which means that there is no need to take it under consideration in this project.

2.1.1 System mockup

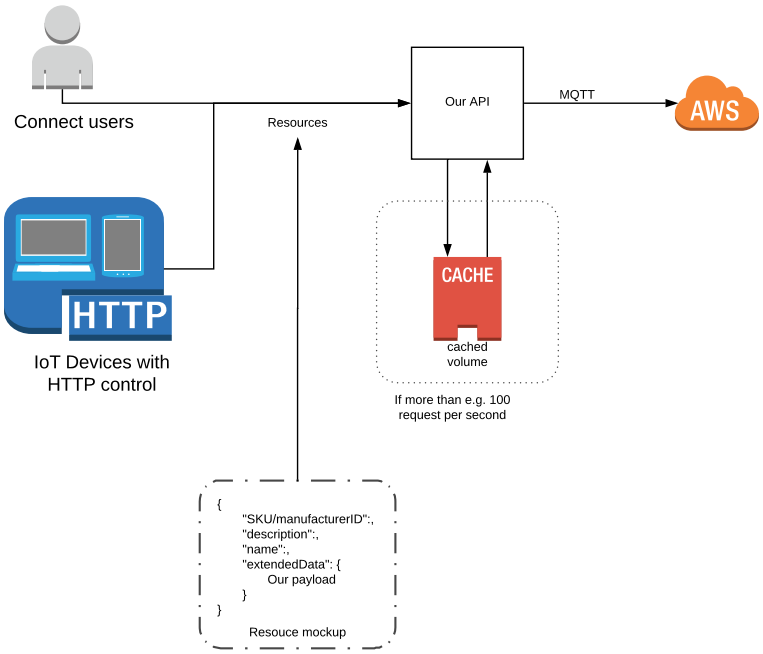


Figure 1: Temporary modelling of our system

3 System development model

3.1 Choice of model

For our software development framework we will be using Scrum. Since we are developing a prototype, new requirements are continuously being added. In conjunction with this and being a small team consisting of 3 people, an agile and iterative software development framework is to be preferred.

3.2 Model description

In agreement with Escio we are using 3 week sprints where Terje (from Escio) is Scrum master. At the end of the sprint we will have a meeting with Escio and have a sprint review where we will discuss the challenges and positives in the finished sprint. Internally we will have Scrum meetings every time we are meeting to keep each other updated on our progress, what we will be working on that day and any impediments we are facing. The backlog will be continuously updated as new requirements will be discovered.

4 Project Organizing and Quality Assurance

4.1 Distribution of responsibilities

Eirik is the project leader. Bartosz is responsible for booking rooms when necessary. Anders task is to communicate with the person responsible for fixing all the permissions we need to the AWS.

4.2 Rules and Procedures

As a rule we have chosen to meet up at least twice a week, as well as have a minimum of 30 work hours per week each. We will have SCRUM meetings when we meet up in order to keep each other updated on the progress that each of us have made.

If a group member decides not to show up to the meeting without notifying the other group members, the penalty is to work up the number of hours that the group meeting lasted. This will be confirmed at the next SCRUM meeting when we will be updating each other.

4.3 Documentation, Standards & Source Code

All choices made during meetings with both the company and the project supervisor regarding the development of the product should be documented and justified.

All written code should follow the chosen language code convention and code that is not self-explanatory should be commented so that it is easier to understand for the other coders. All functions and classes should be documented using either self written documentation tools or by the help of third party tools such as Sphinx. In addition to the documentation there should also be created a wiki so that end-users with little-to-no knowledge about the technology will be able to read and understand how to use the application.

Source code should always be uploaded to Bitbucket. Each coder should have their own branch to work on.

4.4 Testing

The developed prototype should be thoroughly tested with both valid and invalid dummy data. As a prototype it is not as important to have a fully fledged product, but the testing should be done so that we can ensure a smooth transition for further development of the software.

4.5 Risk analysis

Table 1: Risk matrix

	Low	Moderate	Major
Not likely			
Likely			
Most likely			

Table 2: Risk matrix

Risk description	Probability	Impact
Missing access rights	Not likely	Low
Significantly many bugs	Likely	Low
No access to AWS	Not likely	Major
Miscalculated the scope of the project	Likely	Major
Missing the project timeframe	Most likely	Major

4.6 Plan for the management of the main risks

- The company that we are working for should supply AWS account to our use. There can be delays in the delivery process which can further delay our working process and thus delaying the overall project.

5 Implementation Plan

5.1 Gantt Chart

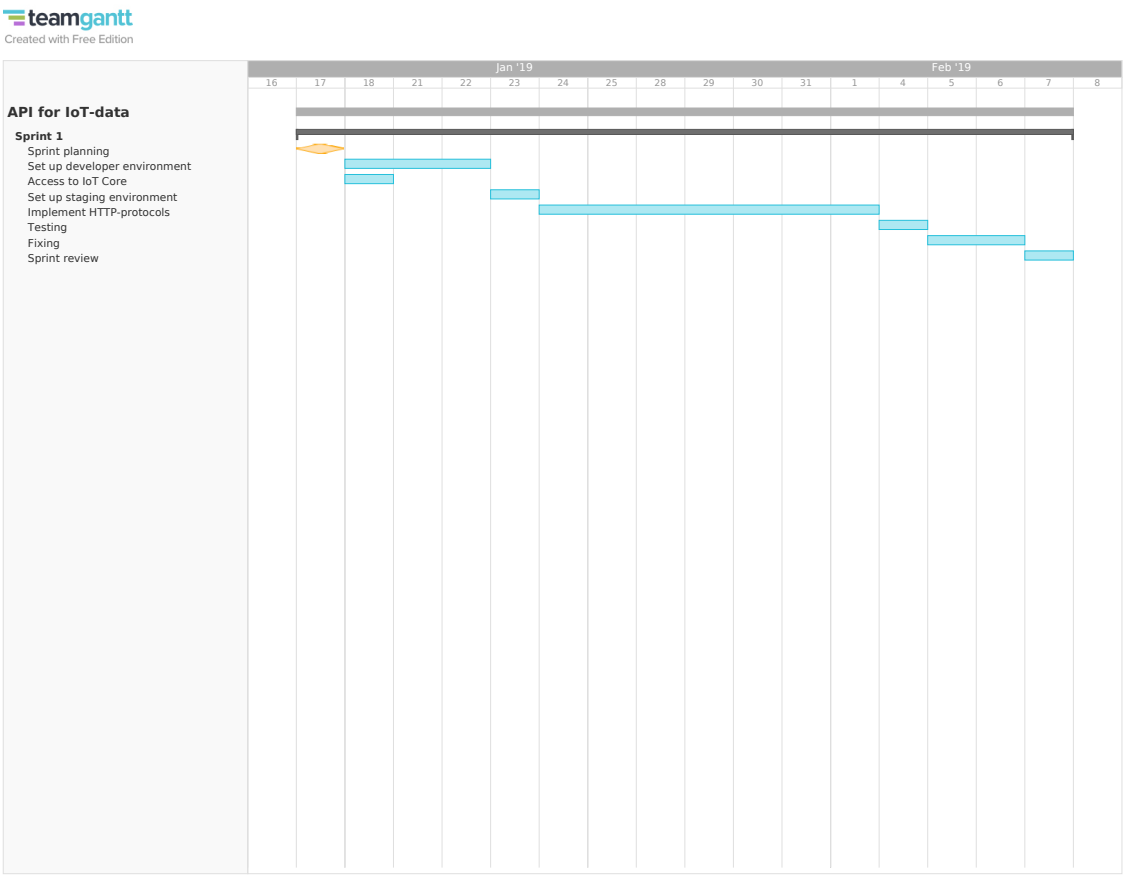


Figure 2: Gant schema for sprint 1

5.2 Milestones and Decisions

We have Scrum sprints that are 3 weeks long. Our milestones will be to be finish with the sprint backlog for each Scrum sprint.

References

[1] Wikipedia contributors. Application programming interface — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Application_programming_interface&oldid=879665721, 2019. [Online; accessed 25-January-2019].

[2] Wikipedia contributors. Internet of things — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Internet_of_things&oldid=880097543, 2019. [Online; accessed 25-January-2019].

[3] Wikipedia contributors. https://simple.wikipedia.org/w/index.php?title=Cloud_computing&oldid=6397437, 2019. [Online; accessed 25-January-2019].

F Project agreement

NTNU

Norges teknisk-naturvitenskapelige universitet

Vår dato Vår referanse

Prosjektavtale

mellom NTNU Fakultet for informasjonsteknologi og elektroteknikk (IE) på Gjøvik (utdanningsinstitusjon), og

TERJE KROGSTAD,

ESCIO AS (oppdragsgiver), og

ANDERS ISAKSEN

BARTOSZ TRACZ

EIRIK PERSSON MAJTEHOLTET (student(er))

Avtalen angir avtalepartenes plikter vedrørende gjennomføring av prosjektet og rettigheter til anvendelse av de resultater som prosjektet frembringer:

1. Studenten(e) skal gjennomføre prosjektet i perioden fra 17. Jan til 02. Mai.

Studentene skal i denne perioden følge en oppsatt fremdriftsplan der NTNU IE på Gjøvik yter veiledning. Oppdragsgiver yter avtalt prosjektbistand til fastsatte tider. Oppdragsgiver stiller til rådighet kunnskap og materiale som er nødvendig for å få gjennomført prosjektet. Det forutsettes at de gitte problemstillinger det arbeides med er aktuelle og på et nivå tilpasset studentenes faglige kunnskaper. Oppdragsgiver plikter på forespørsel fra NTNU å gi en vurdering av prosjektet vederlagsfritt.

2. Kostnadene ved gjennomføringen av prosjektet dekkes på følgende måte:
   - Oppdragsgiver dekker selv gjennomføring av prosjektet når det gjelder f.eks. materiell, telefon/fax, reiser og nødvendig overnatting på steder langt fra NTNU på Gjøvik. Studentene dekker utgifter for ferdigstillelse av prosjektmateriell.
   - Eiendomsretten til eventuell prototyp tilfaller den som har betalt komponenter og materiell mv. som er brukt til prototypen. Dersom det er nødvendig med større og/eller spesielle investeringer for å få gjennomført prosjektet, må det gjøres en egen avtale mellom partene om eventuell kostnadsfordeling og eiendomsrett.

3. NTNU IE på Gjøvik står ikke som garantist for at det oppdragsgiver har bestilt fungerer etter hensikten, ei heller at prosjektet blir fullført. Prosjektet må anses som en eksamensrelatert oppgave som blir bedømt av intern og ekstern sensor. Likevel er det en forpliktelse for utøverne av prosjektet å fullføre dette til avtalte spesifikasjoner, funksjonsnivå og tider.

4. Alle bacheloroppgaver som ikke er klausulert og hvor forfatteren(e) har gitt sitt samtykke til publisering, kan gjøres tilgjengelig via NTNUs institusjonelle arkiv hvis de har skriftlig karakter A, B eller C.

Tilgjengeliggjøring i det åpne arkivet forutsetter avtale om delvis overdragelse av opphavsrett, se «avtale om publisering» (jfr Lov om opphavsrett). Oppdragsgiver og veileder godtar slik offentliggjøring når de signerer denne prosjektavtalen, og må evt. gi skriftlig melding til studenter og instituttleder/fagenhetsleder om de i løpet av prosjektet endrer syn på slik offentliggjøring.

Den totale besvarelsen med tegninger, modeller og apparatur så vel som programlisting, kildekode mv. som inngår som del av eller vedlegg til besvarelsen, kan vederlagsfritt benyttes til undervisnings- og forskningsformål. Besvarelsen, eller vedlegg til den, må ikke nyttes av NTNU til andre formål, og ikke overlates til utenforstående uten etter avtale med de øvrige parter i denne avtalen. Dette gjelder også firmaer hvor ansatte ved NTNU og/eller studenter har interesser.

5. Besvarelsens spesifikasjoner og resultat kan anvendes i oppdragsgivers egen virksomhet. Gjør studenten(e) i sin besvarelse, eller under arbeidet med den, en patentbar oppfinnelse, gjelder i forholdet mellom oppdragsgiver og student(er) bestemmelsene i Lov om retten til oppfinnelser av 17. april 1970, §§ 4-10.

6. Ut over den offentliggjøring som er nevnt i punkt 4 har studenten(e) ikke rett til å publisere sin besvarelse, det være seg helt eller delvis eller som del i annet arbeide, uten samtykke fra oppdragsgiver. Tilsvarende samtykke må foreligge i forholdet mellom student(er) og faglærer/veileder for det materialet som faglærer/veileder stiller til disposisjon.

7. Studenten(e) leverer oppgavebesvarelsen med vedlegg (pdf) i NTNUs elektroniske eksamenssystem. I tillegg leveres ett eksemplar til oppdragsgiver.

8. Denne avtalen utferdiges med ett eksemplar til hver av partene. På vegne av NTNU, IE er det instituttleder/faggruppeleder som godkjenner avtalen.

9. I det enkelte tilfelle kan det inngås egen avtale mellom oppdragsgiver, student(er) og NTNU som regulerer nærmere forhold vedrørende bl.a. eiendomsrett, videre bruk, konfidensialitet, kostnadsdekning og økonomisk utnyttelse av resultatene. Dersom oppdragsgiver og student(er) ønsker en videre eller ny avtale med oppdragsgiver, skjer dette uten NTNU som partner.

10. Når NTNU også opptrer som oppdragsgiver, trer NTNU inn i kontrakten både som utdanningsinstitusjon og som oppdragsgiver.

11. Eventuell uenighet vedrørende forståelse av denne avtale løses ved forhandlinger avtalepartene imellom. Dersom det ikke oppnås enighet, er partene enige om at tvisten løses av voldgift, etter bestemmelsene i tvistemålsloven av 13.8.1915 nr. 6, kapittel 32.

12. Deltakende personer ved prosjektgjennomføringen:

NTNUs veileder (navn): Hao Wang

Oppdragsgivers kontaktperson (navn): Terje Krogstad

Student(er) (signatur): _____ dato 09/01 - 2019

_____ dato 11/1 - 2019

_____ dato 11/1 2019

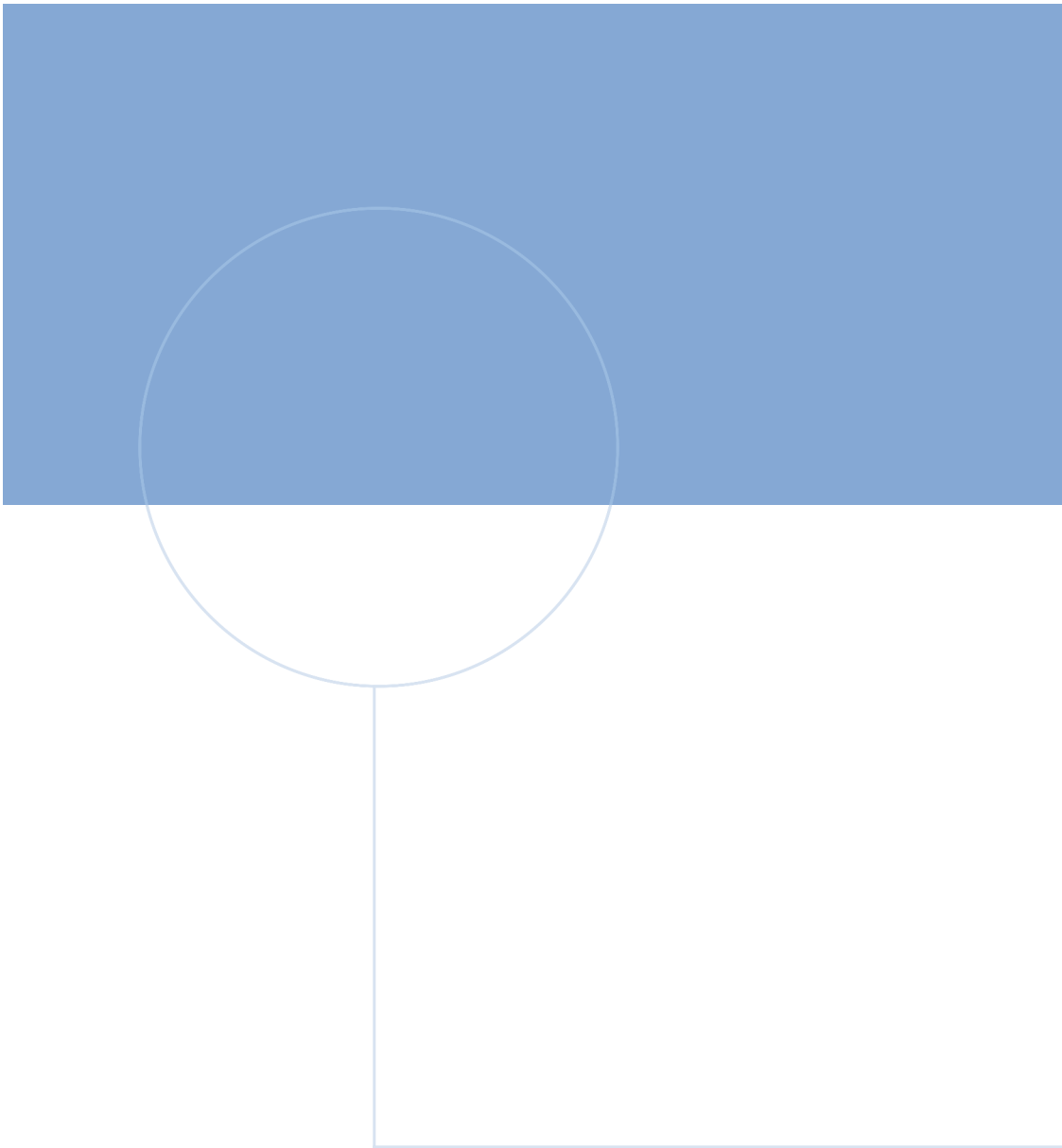
_____ dato 11/1 2019

Oppdragsgiver (signatur): _____ dato _____

Signert avtale leveres digitalt i Blackboard, rom for bacheloroppgaven.
Godkjennes digitalt av instituttleder/faggruppeleder.

Om papirversjon med signatur er ønskelig, må papirversjon leveres til instituttet i tillegg.
Plass for evt sign:

Instituttleder/faggruppeleder (signatur): _____ dato _____



NTNU

Kunnskap for en bedre verden