POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Cloud IoT platform for Access Control

Supervisors

Prof. Giovanni MALNATI

Candidate

Erminio Giuseppe MARINIELLO

March 2022

Abstract

Today's technology has made fields of application that until a few years ago were complex and expensive more and less accessible. Broadband connectivity with a widespread diffusion on the one hand, and refined production processes on the other hand, have exponentially increased the demand for cheaper, more powerful and versatile IoT devices that can be used in many different contexts.

The Internet of Things and Smart Objects have the undeniable advantage to renew and unlock new opportunities from activities that are part of a company's everyday life. A concrete example is the one analysed in this thesis project: the digitalization of a governance process was an opportunity to increase the robustness of the business as well as the driving element of a system that can become the basis of a modern smart-city.

The project involved the development of all the software components of a complete ecosystem: the management cloud platform, the edge software residing on the gateways, a mobile application for field personnel. The strengths of the solution are the expandable and always-connected gateway, the management of multiple sensors and its nationwide presence. These devices are able to collect, process and transmit data to the cloud. Once transmitted the data feeds a database of information that is the real treasure in a world where the data are the real value.

The platform must implement and make the two distinct souls of the project coexist:

- to guarantee protection against unauthorised access to business-sensitive areas, thanks to the implementation of an electromechanical lock and specific sensors
- create a cohesive system between hardware and software that will allow the addition of new sensors for environmental monitoring in the future: all software components and architectural design choices have been made with scalability and high availability in mind.

The main task was to govern a specific business process, in this case the control of access to equipment, cabinets or sensitive company areas: particular attention was therefore paid to implementing specific logics, such as the identification of alarm situations and tampering.

This without precluding the possibility - in the near future - of extending the functionality of these connected objects and make them an active part of a network of probes collecting environmental parameters such as noise or electromagnetic pollution. Nowadays those information have an increasing interests in the context of so-called smart cities.

The project was followed in all its phases: from the realisation of the first prototypes, to the definition of the system specifications in agreement with the client, and the concrete development of the functional components described: the cloud platform, the gateway agent, the mobile application. JavaScript was the common language between all components, from the back-end to the edge software: it therefore represents a proven versatile, effective and powerful technology.

The entire system handles a considerable amount of events on a daily basis without any problems, thanks to its architectural design based on micro-services. Thousands of users operate daily on the more than 30,000 gateways already installed, which will triple by the end of 2022.

Acknowledgements

ACKNOWLEDGMENTS

First of all, I would like to thank Debora, my companion and patient and inexhaustible source of motivation and support. A thought to my family, father, mother and sisters Marialucia and Giovanna for their constant presence.
I would also like to thank my company Ennova, for the many opportunities for professional growth that it continues to offer me.

Table of Contents

List of Tables

List of Figures

Acronyms

ACL

Access Control List

LDAL

Lightweight Directory Access Protocol

IoT

Internet of Things

MERN

MongoDB, Express, React, Node

MEAN

MongoDB, Express, Angular, Node

OTA

Over-the-Air

MIT

Massachusetts Institute of Technology

MVVM

Model-View-Viewmodel

LoRA

Long Range

BLE

Bluetooth Low Energy

GPIO	General-purpose input/output
GUI	Graphical user interface

Chapter 1

Introduction

The Internet of Things (IoT) is a neologism used in the telecommunications sector, a term born out of the need to give a name to real objects connected to the Internet. The term IoT was first used by Kevin Ashton in 1999, a researcher at MIT (Massachusetts Institute of Technology) and is now used to identify any system of physical devices that automatically transfer and receive data over wireless networks.[1]. In fact, we can strictly talk about the Internet of Things when these smart devices are connected in a network: all these sensors became capable to talk each other and exchange information about themselves and their surrounding.

Over the last twenty years, the Internet of Things sector has experienced continuous and unstoppable growth. Broadband connectivity with a widespread diffusion on one hand, and refined production processes on the other hand, has exponentially increased the demand for cheaper, more powerful and versatile IoT devices that can be used in many different contexts: smart home, smart building, smart metering, smart factory, smart city and smart car, as well as the use of complex systems in logistics, agriculture and health, for example.

All these fields of application have in common the (satisfied) need to be able to collect more and more information to make processes monitorable and analysable, so as to make them efficient and sustainable, both economically and environmentally. Internet and smart objects represent the tools capable of connecting business and physical objects, in order to obtain the maximum potential they have to offer.

A smart object can provide us with information about itself such as:

- its operating status: if there are problems, they can be detected and maintenance can be scheduled
- its location: to contextualise the information collected and transmitted in space

They can also tell us something about their surroundings, thanks to their ability to interact with the outside world:

- sensing (e.g. to measure state variables such as temperature, pressure, the presence of fumes or hazardous substances)
- metering (for flow variables such as electricity, water, gas consumption).

An important feature is the ability to process data locally, e.g. to select which information to transmit among those collected or which to use to decide how to actively interact with the surrounding environment, by performing actions. For example, closing a valve or unlocking access to a passageway.

The IoT bridges the gap between physical and digital and allows us to innovate services and create value from the massive amount of data that can be collected and analysed.

Enterprise-wide IoT solutions enable companies to improve their existing business models, but also to create new ones, thanks to access to a pool of accurate, reliable and up-to-date information.

The development of the SmartLock project aimed to combine these two different needs: to support a business process by improving its governance characteristics and, at the same time, to create a widespread network of equipment capable of collecting data of interest for territorial control.

In this specific case, designing a smart device to control access to the client's sensitive infrastructures has a doubly strategic value: on one hand, it strengthens the ability to control assets that are fundamental to a sector such as telecommunications or energy and, on the other, it makes it possible to set up an expandable and widely used network that is ready to be expanded in its functionalities beyond the original one.

In recent years, general attention to issues such as air, magnetic or noise pollution has definitely increased. Particularly in urban areas, it is becoming increasingly common to exceed the legal limits for maximum concentrations of fine dust (like PM2.5 and PM10) in the air, for example.

Exposure to electromagnetic fields is also the subject of much debate. Radio frequency transmissions are an integral part of our daily lives: we all have a smartphone connected to a 4G or 5G data network, home routers, repeaters of all kinds.

Although completely undetectable, electromagnetic pollution is increasing year after year. Having a large number of sensors monitoring the intensity of the fields, to which all citizens are subjected, would make it possible to carry out extensive and continuous monitoring over time. At present, this is a prerogative of a few local initiatives.

Being able to make this information available to organisations and institutions therefore has an extremely high intrinsic value for the whole community.

1.1 Thesis Goals

In this project we tried to combine the need to govern a specific business process, in this case the access control to equipment, cabinets or sensitive business areas, with the possibility to extend the functionality of these connected objects, to make them an active part of a network of devices, for collecting environmental parameters such as acoustic or electromagnetic pollution: the solution is based on an electromechanical lock and an intelligent gateway with integrated sensors, to which a virtually unlimited number of additional sensors can be connected to evolve its functionality.

Given the premises of the previous paragraph, the thesis project developed along two parallel paths. The platform must implement and make the two distinct souls of the project coexist:

- to guarantee protection against unauthorised access to business-sensitive areas, thanks to the implementation of an electromechanical lock and specific sensors
- create a cohesive system between hardware and software that will allow the addition of new sensors for environmental monitoring in the future.

Future expandability is therefore the real differentiating value: all software components and architectural design choices have been made with scalability and high availability in mind.

Chapter 2

Analysis and Design of the solution

2.1 Functional Requirements

As mentioned in the main paragraph, the business requirements of the project involve the development of access control functionality and can be summarised as follows:

- **Management of access permissions**: an ACL (access control list) must be managed for each restricted area to be protected; these lists are based on geographical and time criteria.
- **Historization**: each access attempt (both granted and denied) must be tracked and historicised on a cloud storage component.
- **Identification of break-in attempts and irregular openings**: the gateway is equipped with a series of sensors that enable it to identify break-in attempts and unauthorised openings. The occurrence of such conditions must generate an alarm that must be transmitted for timely management.
- **Mains power failure detection**: a prolonged mains power failure may represent a potential service disruption condition, therefore it must be detected and communicated to the management and monitoring centre with a specific alarm.
- **Collection of environmental data**: the gateway is equipped with various interfaces that allow its expandability for the collection of environmental data such as air, noise or electromagnetic pollution. The widespread distribution of these objects can represent a strategic value.

2.2 System actors

The users of the system are different, and will interface with it differently. Depending on the role, each actor will have to access the information they are responsible for. The main actors are:

- **field operators** users who need to access the reserved areas and must therefore authenticate themselves before access. They interact with the system via a dedicated mobile application
- **installer operators**: users who install the gateways and mechanical components. They interact with the system via a dedicated mobile application with dedicated functionality
- **monitoring team**: users in the operations centre who monitor the system and take charge of any alarms. They interact with the system via a web application
- **administrators**: high privilege users with full visibility among system sections and data

2.3 Platform Architecture

The project is a complete IoT platform, and involves the implementation of the different components that make up such a system.

An IoT platform is the tool needed to manage an entire IoT system. It consists of four primary components: the hardware, the software, the user interfaces and the network. An IoT platform is what connects the four components into a cohesive, manageable, and interpretable system. It helps make data ingestion, communication, device management, and application operations a smooth, unified process.

The diagram shows the main functional blocks of the whole system, which can be grouped as follows:

- **Cloud Platform**: forms the framework on which the whole system is based. It is home to the business logic and data, and ensures consistency of operations. It must be able to grow with the number of devices, so it must guarantee scalability. It provides a front-end with which to access its facilities. All the software is deployed in a Docker Swarm cluster.
- **Embedded Edge Software**: This is the piece of software that runs on the gateways and ensures that they are connected and secure, keeps them up to

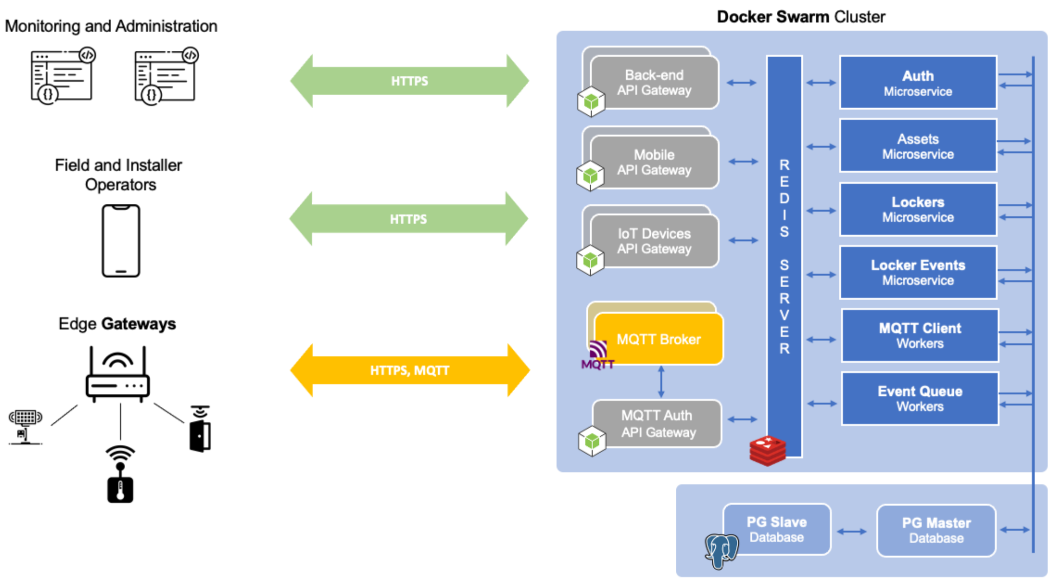


Figure 2.1: Platform Architecture

date and allows them to collect and send data, as well as guaranteeing that all the routines are carried out.

- **Mobile Application**: this is the simplified tool designed for field operators who interface with the system. It communicates with both the gateway and the cloud platform.

2.4 The software stack: Javascript as full-stack language

As we have seen, the platform consists of components that differ in function and composition. There were many different implementation choices, but the need to keep the development phases fast, robust and efficient was strong.

Being initially bound to front end engineering and conceived to be executed within a browser, Javascript has recently made its entrance into the world of runtime environments, revolutionising the way the client and server aspects of an application are built: finally it is possible to use a single widespread programming language to create complex, high-performance applications.

In a very short time, software stacks such as MEAN and MERN have become very popular, gathering support in the developer community. Many variants were born, also thanks to the spread of front-end frameworks like React and VueJS.

According to the popular Stack Overflow portal, Javascript is firmly in the lead as the language with the highest number of active developers in the world: over 12.4 million.

The success is not limited to independent developers or start-ups. Even big names in the world of technology have started using Javascript in their products and services. These include Netflix, PayPal, Airbnb, Medium and many others.

But what are the reasons behind this success?

- **A common language, better development efficiency with fewer resources**
  If all parts of the application are written in Javascript, they can be better understood by all team members: the knowledge gap between back-end and front-end developers, for example, disappears. Team members become cross-functional, allowing a reduction in costs due to better allocation of resources in the project.

- **Performance and Speed**
  Speed of development is matched by high performance. An example is the event-driven, non-blocking IO model used by Node.js that makes it lightweight and fast as compared to other commonly used back-end technologies like Java. Better performance also means fewer resources used, which benefits cost reduction.

- **Extensive code reuse**
  Working with a single language makes it much easier to follow the principle of 'don't repeat yourself' (DRY). Development effort can be reduced by sharing libraries, components and implementation of common logic, for example between front-end and mobile apps. It also improves the general knowledge of the code base.

- **Vibrant community and availability of libraries and toolsets**
  Thanks to the active contribution of giants like Google, Facebook and Microsoft, the community of Javascript developers is probably the largest in the world. Active repositories on Github exceed half a million, and packages available on npm exceed 1.5 million. There are therefore a large number of libraries that can be integrated without licence fees.

Of course, Javascript, like all technologies, is not perfect. In spite of the enormous advantages it can offer, there are drawbacks to pay attention to.
The single process/single thread nature of NodeJS makes it unsuitable for computationally demanding loads, which would keep the event loop busy for too long and prevent it from processing new requests. Fortunately, as the technology has

matured, a solution has been found to overcome this limitation by increasing the parallelism of computational tasks.

Javascript, like all technologies, is not perfect. In spite of the enormous advantages it can offer, there are drawbacks to pay attention to. The single process/single thread nature of NodeJS makes it unsuitable for computationally demanding loads, which would keep the event loop busy for too long and prevent it from processing new requests. Fortunately, maturing technology has found a solution to overcome this limitation: by dividing our software into independent functional blocks that communicate with each other, it is possible to increase the parallelism of computationally intensive tasks without affecting overall performance.

2.5 Technologies and Frameworks of choice

2.5.1 NodeJS as main-runtime

NodeJS is a JavaScript runtime [2], based on the V8 engine developed by Google for its Chrome browser[3].

NodeJS is based on a non-blocking IO event-driven model, which makes it extremely lightweight and efficient. Despite its single process/single thread nature, it manages to serve multiple requests simultaneously, without necessarily having to wait for the previous one to finish its IO operations, such as accessing a file or a database query.

All this is possible because at the base of NodeJS we find the **libuv** library, which allows the asynchronous execution of JavaScript code (entrusted to V8) but, above all, the asynchronous management of the events of the main I/O operations of the kernel, such as TCP and UDP sockets, DNS resolution, file system and file operations, and file system events[4].

NodeJS comes with Node Packager Manager (npm), the world's largest software registry [5]. Using the official npm client (or Yarn as alternative [6]), is possible to download (and distribute) libraries and dependencies to be included is fast and efficient way.

In addition to being able to use a single language for all project components, NodeJS is the ideal choice for real-time and scalable systems: **scalability is baked into the core of Node.js**. It can handle huge number of concurrent connections, and adding more nodes it can scale with the traffic to be managed. This makes it possible to scale the production environment linearly as the number of devices installed in the field grows.

2.5.2 VueJS

VueJS is a progressive open source JavaScript framework designed for the development of simple user interfaces up to sophisticated Single Page Applications (SPA) using the MVVM (model-view-viewmodel) paradigm. [] Created by Evan You, former Google employee and AngularJS collaborator, it is one of the most used frameworks at the moment together with Facebook's React, as well as the most "starred" on Github. The main library is focused only on the visualization layer. The implementation of advanced features is delegated to a set of official libraries designed to manage routing, state or rendering at the server level. A set of compilation tools, a CLI and Devtools for the main browsers complete the package.

The architecture of an application based on VueJS is based on declarative rendering and on the composition of components, isolated units of code with their own reusable business logic. They therefore represent a fundamental abstraction element for building large-scale applications.

A component is essentially an instance of VueJS, directly linking data and DOM (data binding), making both responsive to changes. In VueJS we also find the concept of *Virtual DOM*, which provides that the DOM of the page is not directly updated following a change in the data model, but is mediated through the Virtual DOM. [7]

Compared to its direct competitor React, VueJS can benefit from a **high learning curve**, thanks to a leaner, more elegant syntax and a well-organised structure. There is **no need to use JSX** for the templating part, and the lightness and general performance have made it the framework of choice for the front-end.

2.5.3 Apache Cordova

Apache Cordova (formerly *Phonegap*) is a framework for developing cross-platform mobile applications totally based on HTML, CSS and JavaScript. [8] This framework actually encapsulates a web application within a container built in native technology (Android and iOS). We will therefore speak of hybrid mobile applications, where the rendering of the GUI is entrusted to a webview, but can rely on a bridge capable of executing native code, interfacing directly with the underlying operating system[9].

This approach is certainly slower in terms of performance than an application made entirely with native technologies (Java/Kotlin for Android or Objective C/Swift for iOS), but the continuous progress made by Google and Apple in the development of their operating systems, and the related WebViews that are an integral part, have made the performance gap less and less obvious and therefore penalizing, especially if we look at the important benefits in terms of versatility and speed of development that this platform provides.

As mentioned above, the GUI of an application built with Apache Cordova resides within a webview, but it is possible to use a web framework, such as VueJS, without difficulty. In this way it was possible to share components and part of the business logic between the web front-end and the mobile application, with direct benefits on development time.

2.5.4 PostgreSQL

Postgres is a relational database management system fully compliant with SQL rules and ACID properties. This DBMS is a very versatile solution, as it includes all the tools needed to create modern applications.

In addition to the characteristics of a relational database, Postgres also includes support for JSON-type data - which makes it very similar to a document-based database - together with support for GiST, GIN and B TREE indexes.[10]

The presence of asynchronous notifications makes it possible, if desired, to use Postgres as a messaging system to manage a queue, for example. The NOTIFY and LISTEN commands allow the DBMS to operate as a true pub/sub server, eliminating the need for polling information[11].

Another important aspect is the scalability of the solution: thanks to sharding, it is possible to scale our DBMS horizontally, but the simple partitioning of the tables will already give a big hand in ensuring the best possible performance even in single-node setups.

PostgreSQL has proved to be the right compromise between the rigour of a relational DBMS and the flexibility of a document-based DBMS, thanks to its support for non-relational data formats (Json, XML, Hstore, etc.). It allows complex queries to be executed quickly: PostgreSQL offers performance optimizations including parallelization of read queries, table partitioning, and just-in-time (JIT) compilation of expressions.

2.5.5 Redis

Redis is an open source in-memory data structure store, used as a database, cache, and message broker. It guarantees very high performance as it works with an in-memory dataset. It supports a variety of data types (strings, lists, sets, transactions, Pub/Sub, key TTL) and can be configured in a cluster, ensuring reliability and scalability [12].

Redis is a uniquely versatile tool that has been used both to communicate between the various services of the platform as a message broker and to help manage workloads using multiple queues. As being an associative memory, it has been a useful tool in improving performance as a caching tool for frequently accessed data.

2.5.6 MQTT

The MQTT protocol (acronym of Message Queuing Telemetry Transport) is a transport protocol running over TCP allowing a bi-drectional connections between devices thus maintaining a small networking footprint and low consumption, making it ideal for IoT appliances where bandwidth is limited. [13] Unencrypted connections are used by default, however it is possible to protect the communication channel by using TLS, thus achieving confidentiality of the exchanged data.

This protocol has two main entities, the client and the server - also known as broker - that receives all messages coming from the former and forwards them to the appropriate destination according to defined routing rules. The messages published by the clients are matched to a topic, a string that allows to catalogue in a hierarchical way the information, filter it and forward it to all the clients that have subscribed to the same topic.

The messages published by the clients are matched to a topic, a string that allows to catalogue in a hierarchical way the information, filter it and forward it to all the clients that have subscribed to the same topic.

MQTT provides for three levels of Quality of Service:

- QoS 0: message delivered at most once
- QoS 1: message delivered at least once with acknowledgement
- QoS 2: message delivered exactly once

Higher QoS corresponds to a higher overhead in terms of networking, so it is necessary to adequately weight this parameter according to the type of information to be transmitted to avoid saturating transmissions.

All nodes interested in a particular piece of information can subscribe to the corresponding topics. When a new piece of data is published, all subscribed clients will receive the message containing the data according to the defined QoS.

As mentioned above, topics can be used to define a hierarchy in information. Each topic consists of one or more levels, each separated by a forward slash (topic level separator).

A client interested in receiving messages may subscribe to an exact topic or may subscribe to multiple topics using wildcards. There are two types of wildcards: single-level or multi-level.

As the name suggests, single-level wildcards identified by the plus (+) symbol can be used to replace a single level in a topic. Multi-level wildcards, on the other hand, are identified by the hash mark (#) and allow all topics to be subscribed to from a specific level in the hierarchy.

The broker implementation of choice was EMQX, an Open-Source, Cloud-Native, Distributed MQTT Broker for IoT. [14] The choice was motivated by EMQX's ability to support up to one million connections on a single node: although the expected number of devices is lower, it gave the right guarantees of durability without extension work. It also supports MQTT v3 and v5, authentication with X509 certificates and the possibility of cluster configurations.

2.5.7 Docker and Docker Swarm

Docker is a containerisation technology that enables the creation and use of Linux containers. Docker uses the Linux kernel and its functionality to isolate processes so that they can run independently. The main objective of containers is precisely the ability to run multiple processes and applications separately to make the most of existing infrastructure while retaining the level of security that would be provided by separate systems.

Applications can be enclosed in an image along with all their dependencies to be distributed between different environments. The images can be versioned, making it possible to work with a specific version of our application. Multiple services can be combined, giving developers the ability to recreate an entire infrastructure that mirrors the characteristics of the production environment. Docker also helps to reduce the time needed for deployment: in the past, configuring new hardware took time and investment. Docker containers can be installed in seconds.

When it comes to containerisation, Docker is the de-facto choice. Docker swarm is the official orchestration tool that allows you to manage multiple containers distributed on multiple host machines, raising the level of availability of the entire platform.

The most famous competitor is certainly Google's Kubernetes. Extremely popular on the production systems of the most important systems, it requires greater effort for both creating and maintaining the cluster. Docker Swarm is therefore the preferred solution if you want fast deployment and ease of configuration.

Chapter 3

Cloud Back-end Module

The cloud module is the real heart of the solution and therefore the most critical one. Every element of the project depends on this component, so special emphasis and attention has been given to it.

In order to ensure code **scalability**, **maintainability** and **testability** of code, the principles of **Clean Architecture** have been embraced. It is a software design philosophy that advocates for the separation of layers of code. Each layer is encapsulated by a higher level layer and the only way to communicate between the layers is with The Dependency Rule [15].

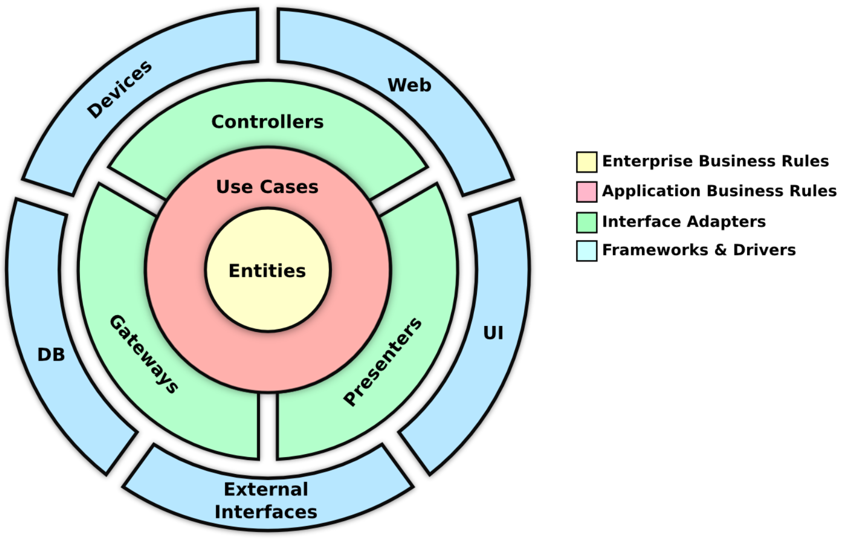


Figure 3.1: The CleanArchitecture [15]

Going from the outermost ring to the inner ring there is is the Server layer which will call and execute Controllers - that can be found in the next layer. The Controllers layer, which will receive the users' data and execute the relevant use cases and return the appropriate response. The Use Cases layer is is where the business logic resides and where it is possible to interact with the Entities Layer. The Dependency Rule states that source code dependencies can only point inwards, meaning each layer can be dependant on the layer beneath it, but never the other way around.

These principles have been reflected in the definition of the following logical components:

- **Controllers** will receive network requests and returns responses from services;
- **Services** will process requests, including validations and third-party reporting, and will save or read data using the repository layer as a source of truth;
- **Repositories** will function as the application's source of truth, will include the DB and external services queries.

3.1 Monolithic vs Microservices architectures

The monolithic architecture can be considered as the classical way to realise applications: we find a large single codebase implementing all functionalities in a software with a low level of modularity. This approach is certainly the most convenients:

- **development** is relatively easier and does not require any special skills;
- **logging** and **debugging** are faster and require less efforts;
- **deployment** can be carried out in a single operation.

However, there are also some negative aspects, which, as we shall see below, have found a solution with microservices

- it is more difficult to manage due the highly tight coupling code;
- as the application grows, start-up times will be longer;
- even a small change requires a deployment of the entire application;
- **scalability**: it is not possible to scale individual functionalities independently
- **resilience**: an unhandled exception can make the entire application unreachable

- **apply a new technology is extremely hard** to because the entire application has to be rewritten with higher impacts and costs.

While a monolithic application is a single coupled unit, m**icro-services architecture breaks it down into a set of smaller independent units**, where all services have their own logic, their own database and runs into a separate process.

According to Sam Newman, "*Microservices are the small services that work together.*" [16].

In fact each component of our application can be understood as an independently deployable unit, which communicates with the others by means of APIs exposed via a lightweight protocol such as HTTP.
This approach to development has undeniable advantages:

- each micro-service has a **single responsibility** and provide a single functionality;
- due to single functionality, they are **smaller** and **easier to understand**;
- when a micro-service gets updated, it is sufficient to deploy only that micro-service and not the whole application;
- **scalability**: if a micro-service is more used than others, it can be scaled horizontally independently of the others;
- **resilience**: if a bug affects one micro-service, it will not affect the other micro-services, so the application will continue to provide the other functionalities to users
- each service can can use the **most appropriate technology based on business requirements**, such as the kind of database used (SQL vs NoSQL);

On the other hand, there are disadvantages which can be summarised as follows:

- complexity: micro-services applications are distributed systems, so connections between modules are needed to ensure inter-communications; externalized configuration, logging, metrics, health checks needed;
- hard to debug because the flow will span between several service on multiple nodes, so extra effort on monitoring/log collector is required
- each micro-service has its independent deployment
- micro-service are costly in terms of network usage and this can lead into into network latency.

The choice between monolithic application or micro-services does not have an absolute correct answer: the choice must be weighed against one's own needs and the capabilities of one's team, but the latter is certainly the ideal choice for a future-proof project.

3.2 Main Functional Services

These listed are the main services implemented by the platform. Each of these runs in a dedicated context, with a dimensioning proportional to the volume of traffic to be managed.

- **Auth Service**: this service is dedicated to the verification and management of authorisation credentials: it queries third-party Active Directory/LDAP services, manages roles and profiles with their respective capabilities;
- **Lockers Service**: implements all business logic related to locker management, from their initialisation to routine activities;
- **Lockers Events**: processes, validates and historizes all events that are entered from the queue by the consumer hooked up to the MQTT broker;
- **Assets Service**: keeps inventories up to date with installations and maintenance carried out over time.
- **MQTT Auth**: implements the MQTT client authentication flow described in section [4.6];
- **MQTT Consumer**: instance of an MQTT client that receives messages published by gateways and places them in a queue towards the Lockers Events service;
- **Event Queues**: this service implements reactive logics when certain events occur, or manages activities scheduled at regular intervals;
- **Push Notifications**: manages outgoing communications to mobile devices, browsers and third-party services.

3.3 The API Gateway

Once the micro-services constituting our application have been identified and developed, it becomes necessary to expose their functionalities to the various clients requesting them. For example, how can the web front-end or mobile application access one or more microservices?

The API gateway pattern is recommended when the microservices-based application has multiple client applications. It acts as distributed system reverse proxy and it is located between clients and underlying micro-services exposing a single entry point to the APIs.

It is possible to have a single API Gateway to which all clients connect. When the number of clients increases and business complexity also increases, this pattern can be dangerous and even become a **single point of failure** of our system. API Gateway it should be segregated based on business boundaries of the client applications and not be a single aggregator for all the internal microservices.

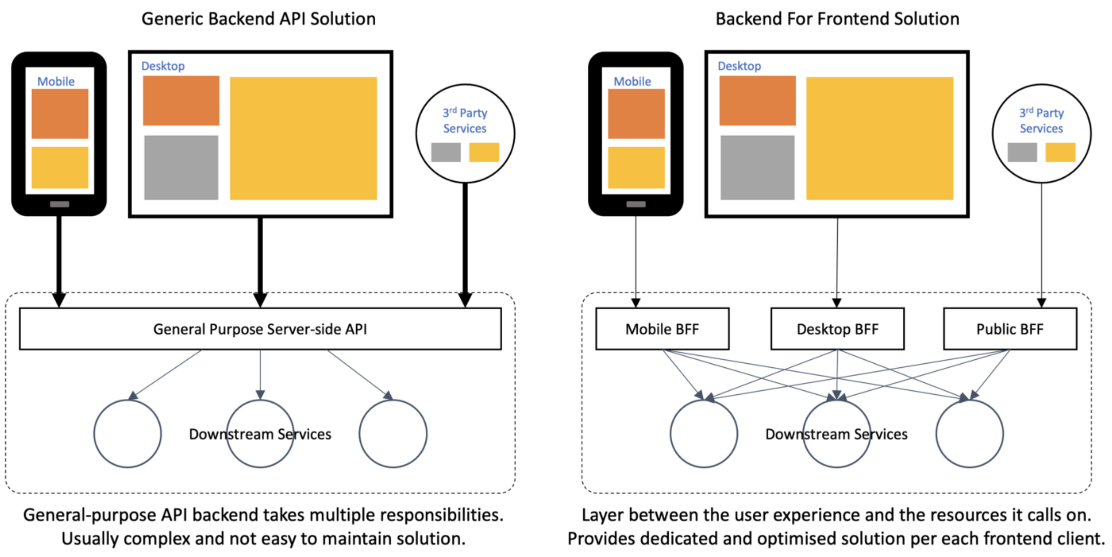

Generic Backend API Solution
Mobile
Desktop
3rd Party Services
General Purpose Server-side API
Downstream Services
General-purpose API backend takes multiple responsibilities.
Usually complex and not easy to maintain solution.
Backend For Frontend Solution
Mobile
Desktop
3rd Party Services
Mobile BFF
Desktop BFF
Public BFF
Downstream Services
Layer between the user experience and the resources it calls on.
Provides dedicated and optimised solution per each frontend client.

Figure 3.2: BFF Pattern [17]

Backend for Frontend (BFF) design pattern is a variant of the API Gateway pattern: instead of a single point of entry, it introduces multiple gateways, one per each client type. Because of that, we can have a tailored API that targets the needs of each client (mobile, web, 3rd parties services, etc..) removing all the bloats caused by keeping it all in one place [18].

API Gateways also fulfil other side functions:

- TLS termination
- Authentication and authorization
- Response caching
- Retry policies and circuit breaker
- Load balancing

- Rate limiting and throttling
- Logging and tracing

As seen in figure 2.1, this is the pattern implemented. There are 4 API gateways identified, for Backend, mobile devices, IoT gateway and MQTT broker respectively. The implementation was done using Express,the unopinionated, minimalist web framework for Node.js.

The Moleculer.JS framwork was used for service intercommunication. It support out-of-the-box all the features needed, such as:

- support for async/await
- request-reply concept
- support event driven architecture with balancing
- built-in service registry and dynamic service discovery
- requests load balancing
- many fault tolerance features
- built-in caching solution
- support for Redis Transporter
- JSON and Msgpack serializers

3.4 Persistence and ORM

To simplify access to the database, it was decided to use the Sequelize library, one of the most well-known and widely used ORMs in the NodeJS ecosystem, which supports Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server. It features solid transaction support, relations, eager and lazy loading, read replication and more. Once the structure of our models and the various relationships have been defined, Sequelize will take care of generating the queries for the DBMS in use.

In accordance with the principles of Clean Architecture, services never call up the database directly. Every model in encapsulated by a Repository.

The **Repository Pattern** has two purposes: first it is an abstraction of the data layer and second it is a way of centralising the handling of the domain objects. The generic repository implements all the CRUD methods (Create, Read, Update and Delete), while the individual implementation defines the exact access criteria.

Chapter 4

IoT Gateway and Edge Software Agent

In the design of an IoT solution, a fundamental role is played by the so-called IoT gateways. They are essentially physical devices that interconnect sensors and other devices to the Internet, but their role is much more complex.

As mentioned above, the gateway is directly connected to the sensors. These can be of different types, using different technologies and protocols (GPIO, Bluetooth, LoRA, ZigBee, etc.) and therefore require a standardisation of information and interfaces.

The data collected also needs to be pre-processed, filtered and aggregated for the implementation of application logics, as well as possibly stored and safely transmitted to the cloud.

The gateway is also responsible for managing Internet connectivity (via SIM) or Ethernet/Wi-Fi adapter, as well as system user interface, configuration management, security and diagnostics. It is clear that this is an essential element of an IoT system. Also in this scenario, NodeJS provided all the necessary tools to implement what was needed.

4.1 The Gateway

This project necessitated the design of customised hardware components. A partner provided an ARM board based on the Freescale i.MX6 SoC and equipped with integrated mobile connectivity, battery backup, Bluetooth interfaces, GPIO, I2, SPI and serial interfaces capable of connecting multiple sensors.

The operating system used is the Yocto Linux Project: the computational resources are not very high, so only the bare essentials have been included. In addition to the drivers and utilities essential in a Linux system, the software solution developed required only the installation of the NodeJS runtime and Redis Server.

4.2 The Agent Software Component

A NodeJS-based application was developed to implement the business logic at the Edge level. Thanks to its event-driven nature and the general optimization of possible resources, it was possible to create a versatile and expandable solution in a very short time.

The application had to meet the following functional requirements:

- Interface with sensors and electromechanical lock.
- Manage the connection with the management server via MQTT to send the collected data and receive commands from the internet network
- Manage through BLE connection the interaction with the mobile application
- Keep in memory the authorization lists and the events not transmitted due to lack of internet connection
- Self updating feature

These four requirements have been implemented in four separate functional modules, executed in independent processes and communicated through the local Redis server. This implementation choice was justified, in addition to the isolation of functionality, by the need to make the individual modules interfaceable by other forthcoming software components.

4.3 Interfacing electro-mechanical lock and sensors

As mentioned in the previous paragraph, the i.MX6 platform on which the gateway is based provides a GPIO interface that allows interfacing with external devices through a series of PINs used as input or output.

The GPIO interface allows the use of signals with two logical levels (high/low). The currents conveyed by the circuitry are very low and therefore the manageable loads are contained, however, by connecting solid-state relays it is possible to control even high power devices such as lights, motors or solenoids. In the same

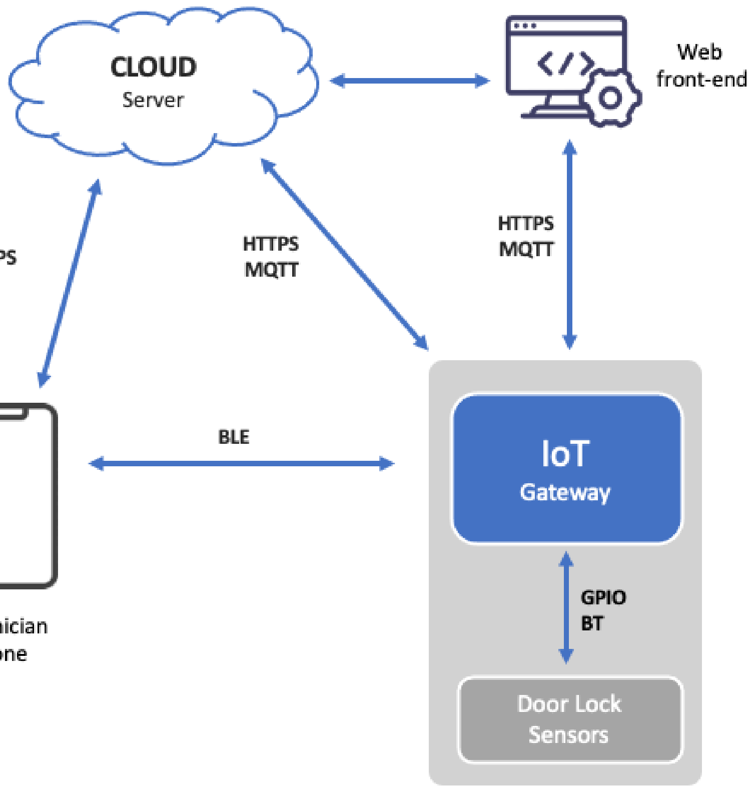


Figure 4.1: Communication scheme

way relays or opto-isolators are used to translate otherwise incompatible signals (like high voltage signals) to the logic signals accepted by GPIO.

To access GPIO there is a framework inside the Linux kernel called gpiolib. This framework provides an API to both device drivers running in kernel space and user space applications [19] . The sysfs interface allows to access GPIO lines and control them reading or writing over specific file description in the */sys/class/gpio* path.

To set-up and control a GPIO pin, is necessary:

1. Export the GPIO writing its number to */sys/class/gpio/export.*
2. Configure the GPIO line writing "out" for output, "in" for input to */sys/-class/gpio/gpioX/direction.*
3. Read value from */sys/class/gpio/gpioX/value* to get current value, or write *1/0* to control PIN value.

NodeJS - like all modern languages - allows access to the file system in an agile and convenient way thanks to the built-in *fs* module. Unfortunately gpiolib and

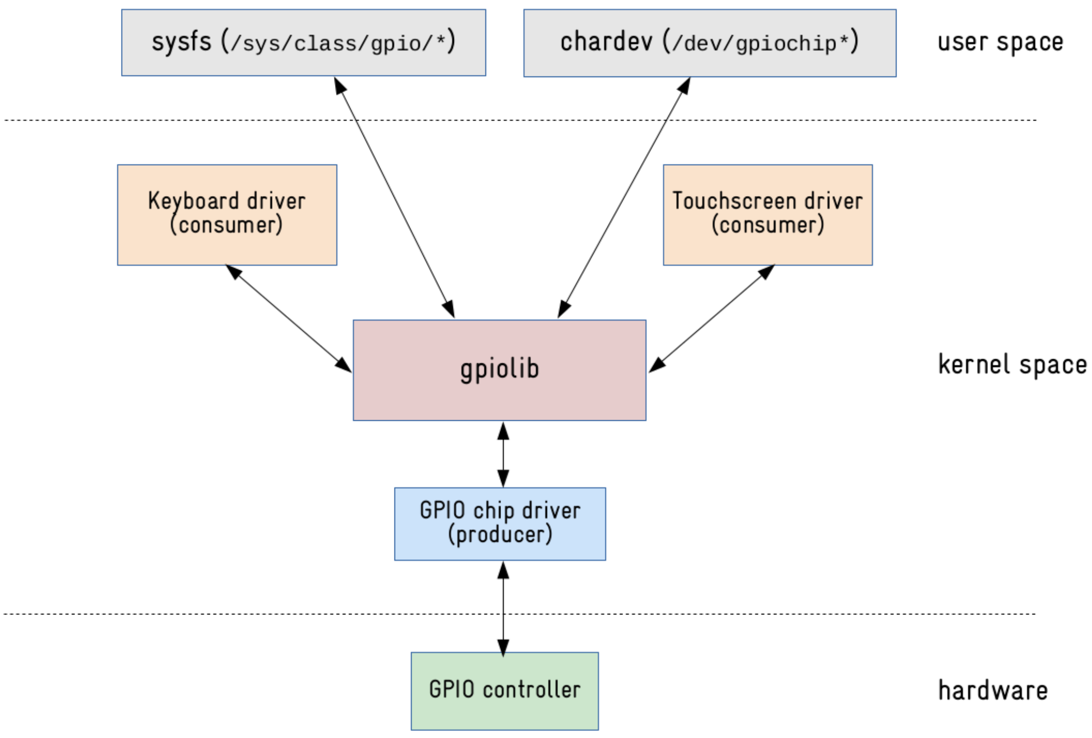

sysfs (/sys/class/gpio/*)
chardev (/dev/gpiochip*)
user space
Keyboard driver (consumer)
Touchscreen driver (consumer)
gpiolib
kernel space
GPIO chip driver (producer)
GPIO controller
hardware

Figure 4.2: Accessing GPIO

the sysfs interface have the big limitation of not having automatic interrupts in case of changes in the input status, so it is necessary to periodically check the descriptor file for changes.

Thanks to a simple function, a PIN status watch function has been implemented, as well as debounce: variations below a threshold level are ignored, eliminating any glitches on the electrical signal or meaningless variations.

```
const checkIfChanged = (key, fd, cb) => {
  const value = this._readInputAsBoolean(fd); // convert to boolean
  const prev = status.get()[key];
  const handler = this.handlers[key];

  if (value === prev) {
    // PIN state back to its original value
    // If a delayed handler is registered, just cancel it

    if (handler && handler.nextValue !== value) {
      clearTimeout(handler.setter);
      delete this.handlers[key];
    }
    return;
  }

  //  a delayed handler already exists, nothing to do
  if (handler && handler.nextValue === value)
    return;

  // registering a new delayed handler
  this.handlers[key] = {
    nextValue: value,
    setter: setTimeout(() => {
      cb(value, status.set(key, value));
    }, WATCH_INTERVAL + WATCH_INTERVAL / 2),
  };
}
```

4.4 Implementing Bluetooth Low Energy (BLE) peripheral

BLE is an alteration of the original Bluetooth technology, designed for low range wireless communications, specifically optimized for low energy connections. It became widespread upon the arrival of so-called wearable devices, such as watches, fitness tracks, blood pressure sensors, all devices that need to preserve battery life to the maximum.

BLE networks must respect a specific topology: each device can operate as central or peripheral respectively.

- **Central**: these are devices such as smartphones or computers with high computational capabilities able to run the software needed to interact with peripheral devices
- **Peripheral**: these are the devices able to collect data and send it to the central device for processing.

A peripheral can only be connected to one central device (such as a mobile phone) at a time, but the central device can be connected to multiple peripherals.

All BLE devices use the **Generic Attribute Profile** (GATT), which defines the way in which data exchanges with each other and intends the concepts of **Profiles**, **Services** and **Characteristics**.

GATT defines the relationship between the Central and the peripheral, assimilating it to the client/server relationship.

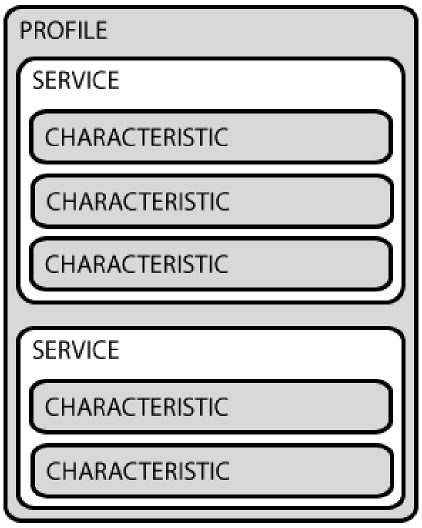


Figure 4.3: Generic Attribute Profile (GATT) Profile

The peripheral is also called a GATT Server, since it retains the definitions of profiles, services and features. The central device (the smartphone for example) is instead the GATT Client that sends requests to the server.

Whenever the two devices need to exchange data, a **GATT Transaction** is created, initiated by the primary device (the GATT Client) which receives responses from the secondary device, the GATT Server.[20]

A Profile is basically a pre-defined collection of Services depending on the kind of the device itself, like a Heart Rate Monitor. Services are used to group individual data - represented by characteristics - into logical entities. Both services and features are identified by a 16-bit (for officially adopted BLE services) or 128-bit (for custom services) UUID.

Characteristics are the point of interaction with the BLE peripheral: each characteristic is associated with a type of information and defines its access mode: **read**, **write**, **notifications** to receive an update every time the value of a characteristic changes.

The library chosen for this part is bleno, which offers a developer-friendly interface to Bluetooth HCI sockets.

Defining a Service is quite simple:

```
/*** service.js ***/
const bleno = require('bleno');
const { PrimaryService } = bleno;

const characteristic = require('./characteristic.js');

const service = new PrimaryService({
    uuid: 'fffffffffffffffffffffffffffffff0', // or 'fff0' for 16-bit
    characteristics: [
        characteristic
    ]
});
module.exports = service;
```

The Characteristic will implement handlers for the 3 properties used:

- **read** to get current status payload
- **write** to send commands to the device
- **notify** to register a callback used to send device updated data as soon as they are available. In order to prevent flooding the device with no updated values, the last sent packet is kept in memory to compare it with the next iteration.

```
/*** characteristic.js ***/
const bleno = require("@abandonware/bleno");
const config = require("../../config");
const logger = require("../../helpers/logger");

const { Characteristic } = bleno;

const status = require("../status");
const commandParser = require("./commands");

const data = {
  payload: null
};

module.exports = new Characteristic({
  uuid: "fffffffffffffffffffffffffffffff1",
  properties: ["read", "write", "notify"],
  secure: ["read", "write", "notify"],
  value: null,

  onReadRequest: async function (offset, callback) {
    logger.info("Read status Request");

    callback(
      this.RESULT_SUCCESS,
      Buffer.from(Buffer.from(status.getPayload()))
    );
  },

  onWriteRequest: async (data, offset, withoutResponse, callback) =>
   {
    try {
      // parsing command
      const response = await commandParser.parse(data.toString("utf-8
   "));

      // sending ack
      callback(Characteristic.RESULT_SUCCESS);

      if (this.sendDataCallBack && response) {
        this.sendDataCallBack(Buffer.from(`iR${response}$`));
      }
    } catch (err) {
      console.warn("Error in onWriteRequest", err.message);
      callback(Characteristic.RESULT_UNLIKELY_ERROR);
    }
  },
  onSubscribe: (maxValueSize, callback) => {
    logger.info("New Subscription to BLE");

    // saving callback for further usage
    this.sendDataCallBack = callback;

    data.payload = null;

    this.sensorsNotification = setInterval(async () => {
        // retrieve current status payload
      const payload = status.getPayload();

      // send payload to device if updated
      if (data.payload !== payload) {
        data.payload = payload;
        callback(Buffer.from(payload));
      }
    }, 1 * 500);
  },
  onUnsubscribe: () => {
    logger.info("Device unsubscribed");

    // clearing callback and interval
    this.sendDataCallBack = null;
    clearInterval(this.sensorsNotification);
  },
});
```

4.5 Securing BLE Connection and Data Exchanges

As you can see in the previous code block, Bleno allows you to specify the **secure** attribute for one or all of the Characteristic properties.

When a BLE connection is established, the **Pairing** process happens in two to three phases: in the first phase devices exchange basic information about their capabilities and what they can do. This exchange is not encrypted. In the next step, each device generates and exchanges the necessary keys to prevent tampering with the connection. The third (and optional) phase is called **Bonding**: each device stores the authentication data they exchanged during pairing in order to use them for future connections. [21]

As we can guess, the second phase is the most vulnerable and critical phase of the process. To avoid security problems, there are two types of BLE connections that can be established: **Legacy** and **Secure**.

Legacy connections can be implemented for BLE versions 4.0, 4.1 and 4.2. During the process, devices exchange a value called a Temporary Key (TK) and use it to generate a Short Term Key (STK), which is then used to authorize the connection. Legacy BLE connections are insecure by default. Secure connections were introduced with BLE 4.2 and are not backward compatible with BLE versions lower than 4.2. Connections implement the Elliptic-curve Diffie-Hellman (ECDH) algorithm for key generation and introduce a more complex key authentication process. Connections can benefit from default protection from passive eavesdropping.[22]

Although the communication channel can be considered secure, it was deemed appropriate to add an additional layer of protection at the application level. Every time the gateway connects to the cloud management system, it generates a new 16-byte key and an initialization vector of the same length.

All data exchanged between gateway and smartphone will be encrypted using AES-128-CBC algorithm. Like the gateway, also mobile devices will obtain - after proper authentication and authorization - the same keys, thus enabling the clear reading of exchanged data. At the end of each connection, the gateway will request a new key from the server, which will invalidate the previous one.

4.6 Connection with Cloud: MQTT Client

To enable a two-way connection with the cloud server, the software integrates an MQTT client based on the Mqtt.js library, fully written in Javascript.

After the initial configuration and all subsequent connections, the gateway calls the auth webservice that returns an identifier and a JWT token. This information will be used to authenticate the gateway to the MQTT broker.

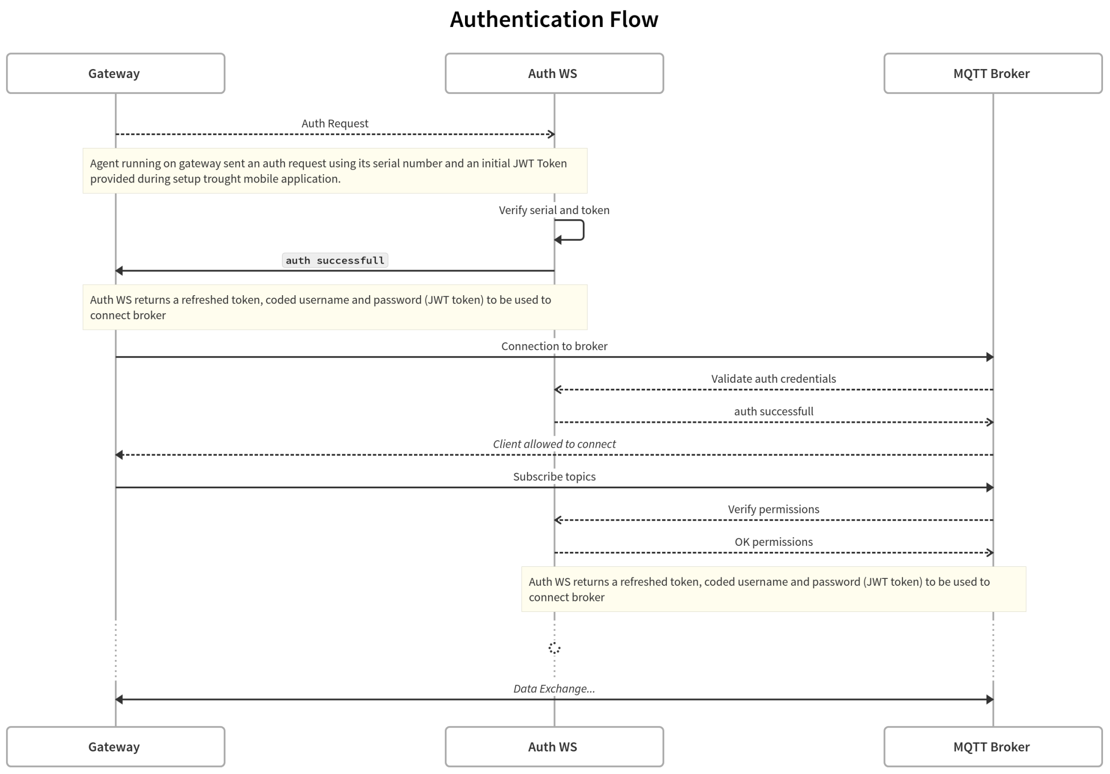

Authentication Flow
Gateway
Auth WS
MQTT Broker
Auth Request
Agent running on gateway sent an auth request using its serial number and an initial JWT Token provided during setup trought mobile application.
Verify serial and token
auth successfull
Auth WS returns a refreshed token, coded username and password (JWT token) to be used to connect broker
Connection to broker
Validate auth credentials
auth successfull
Client allowed to connect
Subscribe topics
Verify permissions
OK permissions
Auth WS returns a refreshed token, coded username and password (JWT token) to be used to connect broker
Data Exchange...
Gateway
Auth WS
MQTT Broker

Figure 4.4: MQTT Authentication Flow

The MQTT Client library of choice in the Javascript ecosystem is MQTT.js. This provided all the necessary things: support with MQTT 5 protocol, automatic reconnection management, customizable keep alive timeouts, support for TLS and client certificates.

In order to manage in a centralized way the connection with the cloud server, a special class has been implemented that implements the entire flow and remains accessible from any point of the application using the Singleton pattern. The competencies of this class can be summarized as follows:

- Authenticate the gateway on the cloud system
- Manage the connection to the MQTT broker
- Manage the commands that may come from it, sending the results of the query
- Maintain in a buffer eventual packets not sent due to absence of connectivity

For this last point, the **NeDB** file-based database was used. Again, the main requirement was lightness and NeDB provided the right balance between functionality and memory requirements. When a software module wants to send data to the broker, if the gateway is off-line the method will write a row to the internal database. As soon as the connection is restored, all the database contents are emptied and sent to the server.

In order to maximize the consumption of data traffic, the payload of the data exchanged on the broker has been serialized using not the classic JSON, but the **MessagePack** binary format, the same used in popular open-source projects like Redis or Fluentd, in its pure Javascript implementation **msgpack5**. [23]

Converting from JSON to MessagePack and vice versa is quite simple, using the proper methods provided by the library. Even with small objects the improvement is appreciable, as you can see from the example.

The difference between the object serialized with JSON.stringify and MessagePack is almost 30%: 54 bytes in the first case, 39 in the second one.

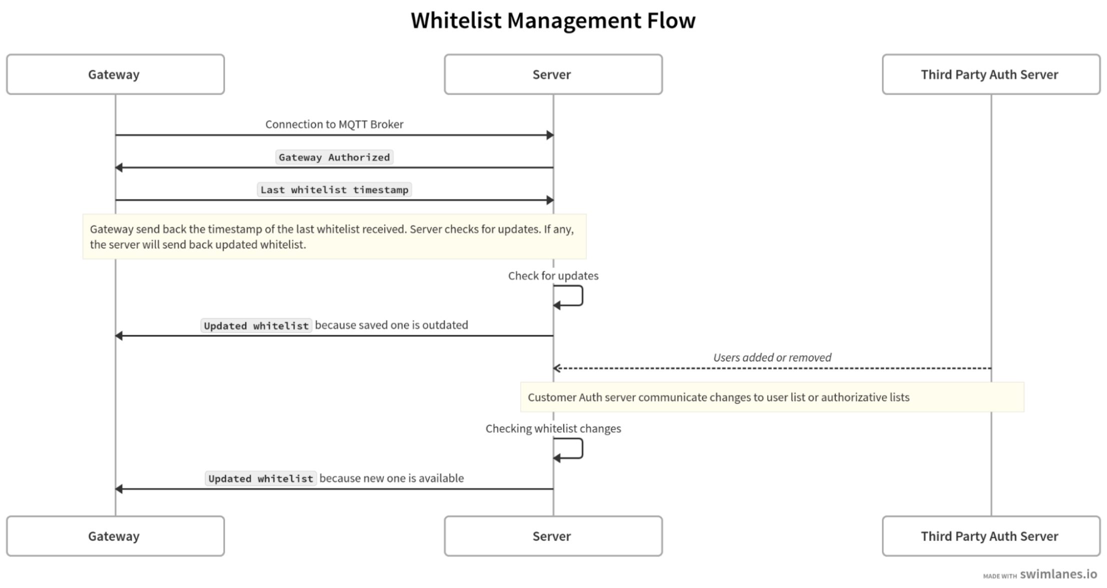
```
const { encode, decode } = require('msgpack5')();

const myJsonPayload = {
    data: 'foo',
    value: 'bar',
    timestamp: Date.now()
};

const stringify = JSON.stringify(myJsonPayload);
const encoded = encode(myJsonPayload);

// prints "Object as String length = 54"
console.log('Object as String length = ' + stringify.length)

//  prints "Encoded Payload Length = 39"
console.log('Encoded Payload Length = ' + encoded.length);
```

The encode function returns a Buffer that can be directly published to the broker using the publish method of the MQTT client instance.

4.7 Managing Authorized Users

The functionality around which the whole system revolves is that of access control. The gateway is connected to an electromechanical lock placed inside cabinets which only authorized users can access. The opening requests are sent through the mobile application with which all interested parties are equipped.

However, not all users are authorized to open any cabinet at any time of the day. In fact, there are regularly updated authorization lists which are then distributed by the management system to the gateways which receive the list and save it internally so that this list is always available even if the gateway is off-line.



Gateway send back the timestamp of the last whitelist received. Server checks for updates. If any, the server will send back updated whitelist.

Customer Auth server communicate changes to user list or authorizative lists

Figure 4.5: Whitelist Management Flow

As soon as the gateway successfully establishes a connection to the management server through the MQTT protocol it sends the timestamp of the last received whitelist. If a more updated revision is available on the server, it will be sent to the gateway which will store it.

The maintenance of the user list is the responsibility of an external system integrated with the platform through APIs. When changes are applied, the management server recalculates the new lists for each impacted gateway. If there are changes, the new whitelist is sent to the gateways through a message published on the topic associated with the gateway.

Once received, internally the whitelist is saved within a collection of the integrated NeDB database and held in persistence as well as in memory for quick access to the information.

4.8 Auto-updating feature

Connected gateways are expandable by nature, and so has to be the software. Since NodeJS is an interpreted language, updating the agent is as simple as updating the files in the directory. At each software startup - at regular intervals - the agent makes a call to a web-service that checks for new available versions, along with the URL to download it from and its signed digest to ensure that only authentic packages get installed on the gateway.

At each release, a script automatically creates a .tar.gz archive containing the delta of files updated since the previous release. Then the SHA-256 digest of the same file is calculated and signed using RSA encryption with a private key.

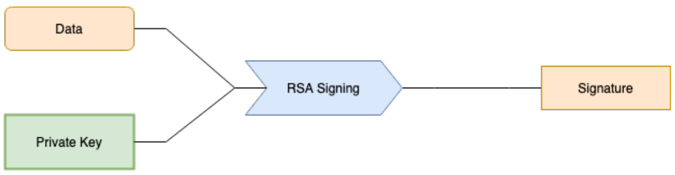


Figure 4.6: OTA Update Archive Signature

When the gateway queries the server to check for new updates, the URL of the package and its digital signature are returned. The file is downloaded, the bash recalculated, and the signature verified through the reverse process of public key decryption. If the two hashes differ, the update is discarded and deleted.

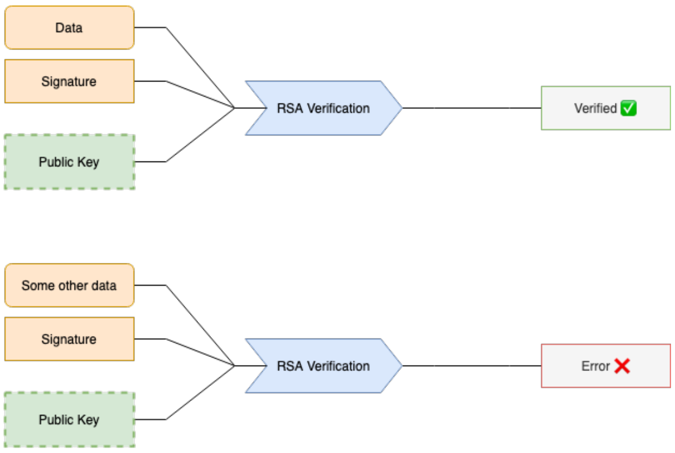


Figure 4.7: OTA Update Archive Verification

Chapter 5

Mobile Application

The largest group of users of the platform is made up of operators who move around the area and go to the various cabinets located throughout the territory for periodic maintenance activities.

The tool considered most functional for them is certainly a mobile application that makes it feasible to exploit the versatility and convenience of the smartphone that everyone has.

The emphasis was mainly on the usability and ease of use of the application, especially considering the audience not too expert in computerized tools.

There are three main functionalities that the application must satisfy:

- connection to the gateway for opening the lock
- initialization of new gateways to be installed or replaced
- receiving notifications and alerts

Obviously, in order to access the application it is necessary to have appropriate login credentials and access to the different features will depend on the role associated with the user.

For this project it was decided to opt for a hybrid app based on the Apache Cordova framework. The main drivers of this choice can be summarized in these points:

- significantly **shorter development time**: since all the GUI is done in HTML and Javascript, this part can be totally shared between the Android and iOS app. The use of VueJS combined with Vuetify has made the experience satisfying and fluid, as well as perfectly aligned with Google's Material Design specifications.

It is also possible to reuse and share libraries originally written for the Agent or the front-end and vice versa.

- use of **native features**: thanks to the plugins offered by the community, the integration of features such as Bluetooth, locations and maps, fingerprint reader and bar-code required limited adoption efforts

- **Over-The-Air updates**: native code and static JS assets can be updated independently. Leveraging the Codepush service in Microsoft's Appcenter suite, it was possible to distribute updates out of the store, allowing small changes or fixes to be distributed almost in real time across the entire user base.

5.1 Anatomy of Application

As mentioned earlier, this module basically started from a VueJS project. The core of the application is in fact developed in JavaScript, and once compiled will produce a set of static assets that will be encapsulated within our native app.

A wide ecosystem has been created around VueJS, with many community contributions that have extended functionality by drawing on other popular projects, such as in our case Apache Cordova. Thanks to the **vue-cli-plugin-cordova** plugin, in fact, it was possible to integrate the functionalities of the previously mentioned framework and at the same time to apply the necessary changes to guarantee its correct functioning, such as configuring the router and the paths where to place the files.

Below are the main plugins used in the implementation of the native application:

- **cordova-plugin-ble-central**: this plugin enables communication between a phone and our Bluetooth Low Energy (BLE) peripheral implemented through the gateway agent

- **cordova-plugin-fcm-with-dependecy-updated**: this plugin simply integrate the Firebase Cloud Messaging features in order to allow app to receive push notifications

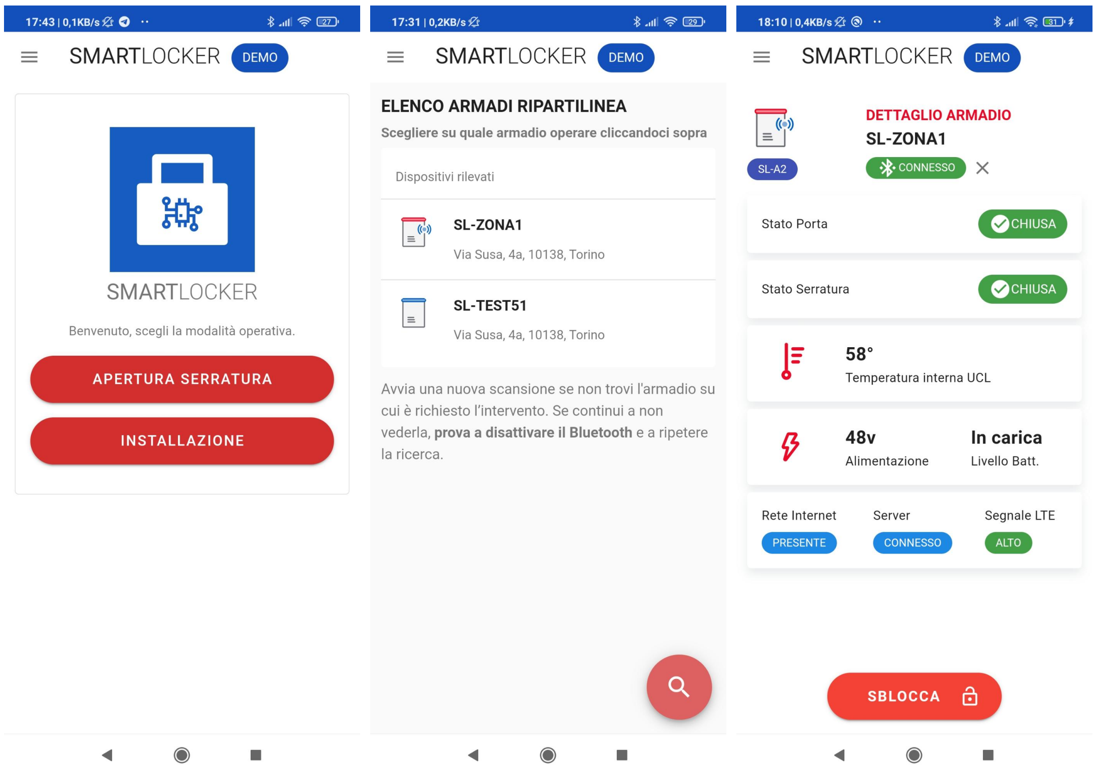
- **cordova-plugin-code-push**: this plugin coming from Microsoft brings the dynamic client update capability through OTA

- **cordova-plugin-secure-storage**: sensitive data as tokens are stored in encrypted way on the device. This plugin was combined with vuex-persiststate in order to avoid saving in local storage

- **phonegap-plugin-barcodescanner**: as the name suggests, with this plugin you can capture barcodes or QR codes using your smartphone camera
- **cordova-plugin-nativegeocoder**: with this plugin it is possible to convert a location into a full address and vice versa.

5.2 Access to nearby cabinets

The main user of this system in terms of numbers is definitely the field operator who goes to the cabinets for routine activities. Operators can open only the cabinets in their immediate vicinity so all interaction will take place using the BLE connection.



Benvenuto, scegli la modalità operativa.

ELENCO ARMADI RIPARTILINEA
Scegliere su quale armadio operare cliccandoci sopra

Dispositivi rilevati

SL-ZONA1
Via Susa, 4a, 10138, Torino

SL-TEST51
Via Susa, 4a, 10138, Torino

Avvia una nuova scansione se non trovi l'armadio su cui è richiesto l'intervento. Se continui a non vederla, **prova a disattivare il Bluetooth** e a ripetere la ricerca.

DETTAGLIO ARMADIO
SL-ZONA1

Stato Porta: CHIUSA
Stato Serratura: CHIUSA
58° Temperatura interna UCL
48v Alimentazione
In carica Livello Batt.
Rete Internet: PRESENTE
Server: CONNESSO
Segnale LTE: ALTO

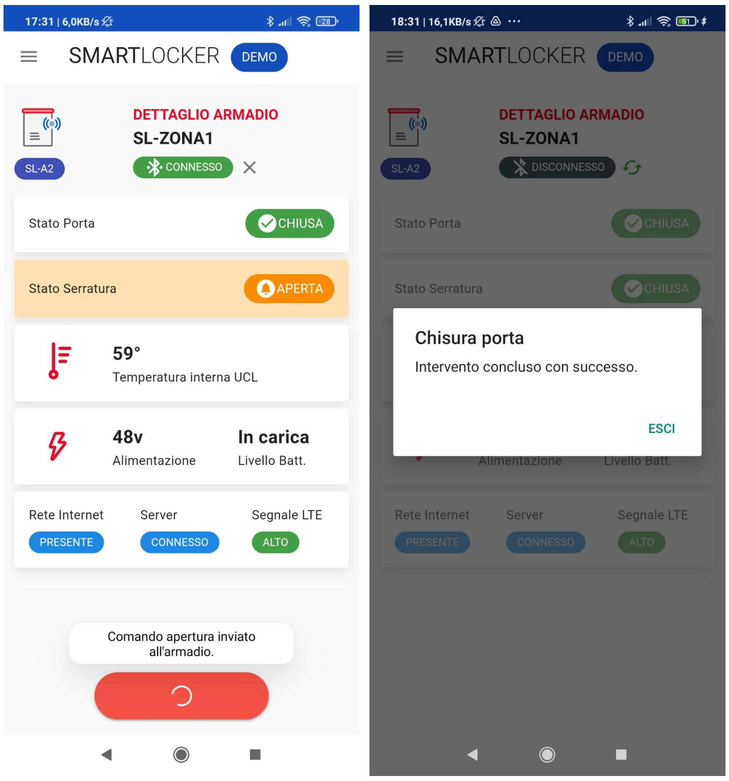
Figure 5.1: Scan and connection to nearby Gateway

The screenshots above illustrate the main flow from the welcome screen to the details tab of a gateway. The scan shows all detected devices that are exposing a specific service and that comply with the naming convention. Whenever a gateway is recognized, human-readable information is downloaded from the server to enrich the list with useful information.

The operator can quickly consult some information related to the functioning of the gateway:

- internal temperature
- presence of power supply
- battery level (if no power supply is detected)
- type (2G, 3G, LTE) and strength of the mobile network signal
- connection to the data network and cloud server

Once connected, opening the door is as simple as pressing the Unlock button. The application will generate an opening request containing information about the user and the current timestamp, it will be encrypted with AES and sent to the gateway. If the packet is valid and the user is found to be valid and authorized, then the gateway will send the unlock signal to the lock. If the lock is not opened within 10 seconds, the operation will be cancelled and the lock will be locked again.



Comando apertura inviato all'armadio.

Chisura porta

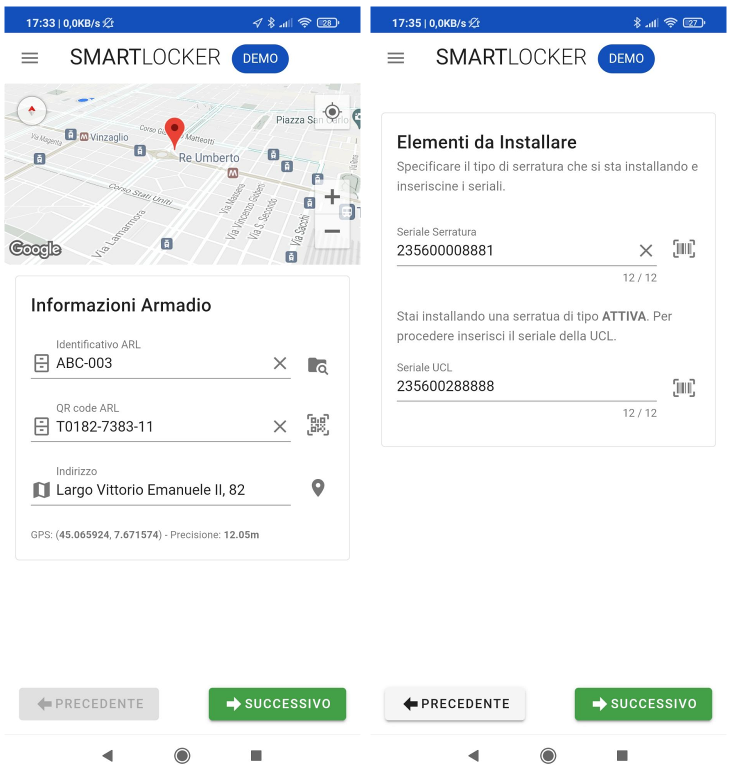
Intervento concluso con successo.

Figure 5.2: Open door and close message

To prevent locks from being inadvertently left open, the platform implements an alert mechanism: if an unlock request is made but the user wanders out of BLE coverage range, the gateway will send a message to the cloud management server, which then sends a push notification to the device and informs of the forgetfulness.

5.3 Initialization of new gateways

In order to operate properly, each gateway must be surveyed at a specific location and pass a testing procedure that attests to the proper assembly of components. A wizard has been implemented directly in the mobile application.



Informazioni Armadio

Identificativo ARL
ABC-003

QR code ARL
T0182-7383-11

Indirizzo
Largo Vittorio Emanuele II, 82

GPS: (**45.065924, 7.671574**) - Precisione: **12.05m**

Elementi da Installare

Specificare il tipo di serratura che si sta installando e inseriscine i seriali.

Seriale Serratura
235600008881
12 / 12

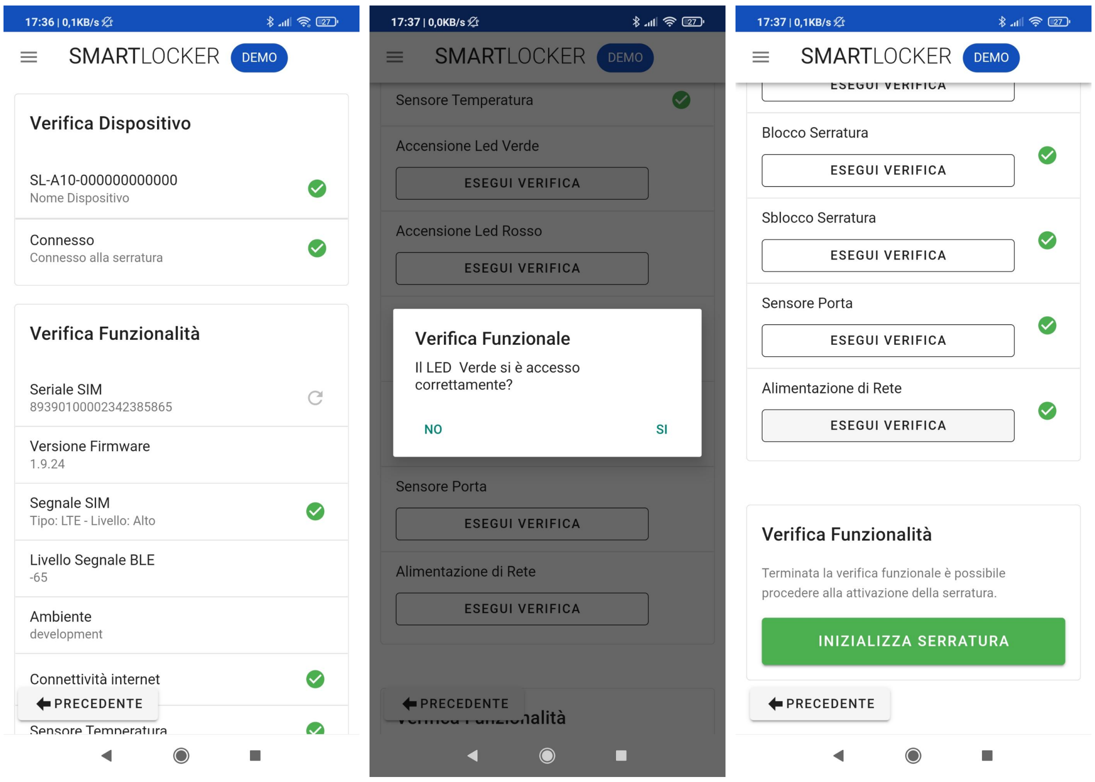
Stai installando una serratua di tipo **ATTIVA**. Per procedere inserisci il seriale della UCL.

Seriale UCL
235600288888
12 / 12

Figure 5.3: Gateway initialization Wizard

The first part of the process involves acquiring the current location: along with latitude and longitude, the full address is also read through the internal Geocoder built into Android. The QR-Code of the installation site and the barcodes (EAN13 encoded) of the SKUs of the installed parts, on the other hand, can be read through the barcode reader integrated in the application.

The following part, on the other hand, foresees the acquisition of preliminary information (such as the reading of the SIM serial as well as a check of the mobile


17:36 | 0,1KB/s

SMARTLOCKER DEMO

Verifica Dispositivo

SL-A10-000000000000
Nome Dispositivo ✓

Connesso
Connesso alla serratura ✓

Verifica Funzionalità

Seriale SIM
89390100002342385865

Versione Firmware
1.9.24

Segnale SIM
Tipo: LTE - Livello: Alto ✓

Livello Segnale BLE
-65

Ambiente
development

Connettività internet ✓

PRECEDENTE

Sensore Temperatura ✓

17:37 | 0,0KB/s

SMARTLOCKER DEMO

Sensore Temperatura ✓

Accensione Led Verde
ESEGUI VERIFICA

Accensione Led Rosso
ESEGUI VERIFICA

Verifica Funzionale
Il LED Verde si è accesso correttamente?
NO SI

Sensore Porta
ESEGUI VERIFICA

Alimentazione di Rete
ESEGUI VERIFICA

PRECEDENTE

Verifica Funzionalità

17:37 | 0,1KB/s

SMARTLOCKER DEMO

ESEGUI VERIFICA

Blocco Serratura
ESEGUI VERIFICA ✓

Sblocco Serratura
ESEGUI VERIFICA ✓

Sensore Porta
ESEGUI VERIFICA ✓

Alimentazione di Rete
ESEGUI VERIFICA ✓

Verifica Funzionalità

Terminata la verifica funzionale è possibile procedere alla attivazione della serratura.

INIZIALIZZA SERRATURA

PRECEDENTE

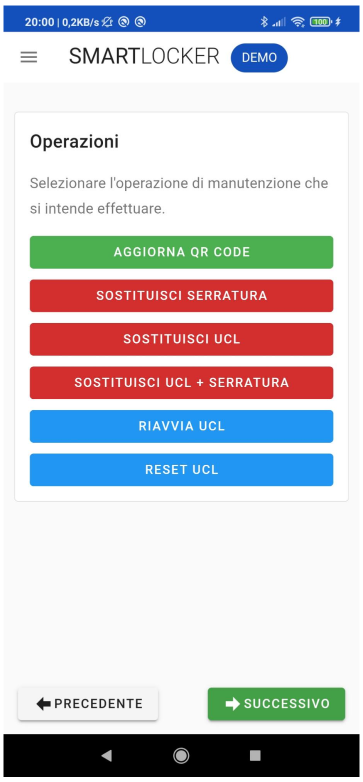
Figure 5.4: Gateway self-diagnostic tests

network signal quality and the version of the agent running on the gateway) and the testing of the functionality of the main components: LEDs, lock release and lock, verification of door and power sensors.

Each test is interactive, requiring the installer to provide confirmation that the output is as expected. Only when all tests have given a positive feedback, it is possible to proceed to the actual initialization phase. The captured data is sent to the management server to generate the provisioning data. It will be the mobile application that will send them via BLE to the gateway. Once the data exchange is finished, the gateway is restarted and the procedure can be considered completed.

A very similar procedure is designed to allow the replacement of one or more previously installed components. In addition to the latter it is possible to launch a total reboot of the gateway or a reset to factory settings.

The similarity between the installation and replacement processes made it possible to reuse many of the components made for the former in the latter as well. The versatility of VueJS was thus able to express itself completely.



Operazioni

Selezionare l'operazione di manutenzione che si intende effettuare.

Figure 5.5: Gateway maintenance features

5.4 Creating OTA Updates

Generally, when you need to distribute an update to an application, you need to compile the source code, generate an .apk file (in the case of Android) or an .ipa file (in the case of iOS), sign it with developer certificates and upload it to the platform's app store.

Apart from the technical time required to produce the updated file, once submitted it can take several hours (or in some cases days) before it is validated and officially published. Only at this point the users of our application will be able to download the updated version on their device. So what if you need to distribute an urgent patch or a hotfix to the whole installed base? Services like AppCenter CodePush by Microsoft [24] are born to solve this specific need.

Each time you launch the app, you can query Code Push to see if there are any new updates that you can install. If available, it is downloaded and installed, regardless of the updates available on the Store.

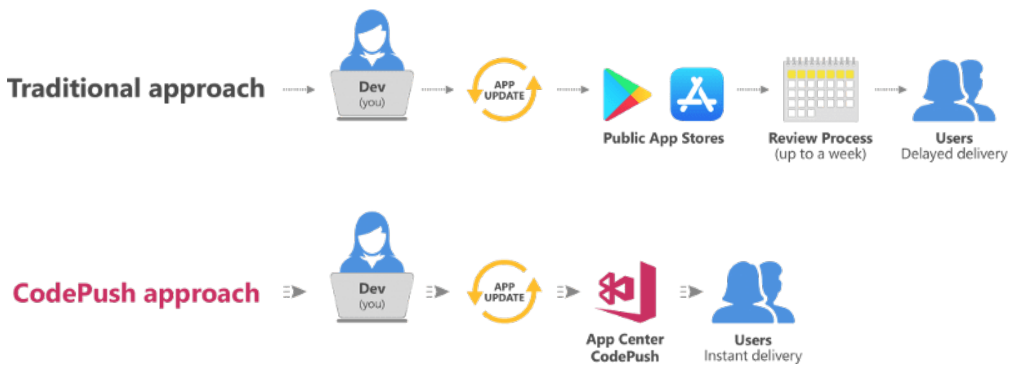


Figure 5.6: Appcenter CodePush

However, this approach has limitations: an OTA update can only concern the GUI part (the one made in Javascript) and not the native part. If you need to update a library or add a new plugin, you must go through a submission to the store. Each update package is therefore tied to the version of the binary for which it was compiled. It is possible in any case to intervene on the target binary version, thus having more control over which updates each binary can receive without having compatibility problems.

If desired to test a new feature on only a portion of your user base, CodePush allows you to specify a roll-out percentage, or conversely mark an update as mandatory for everyone.

Chapter 6

Optimizing time series DB with PostgreSQL

A typical IoT solution scenario involves data from devices being historicized and accessed using time as the primary criteria. Depending on how many devices are on the system and how often they are updated, the size of these databases can grow significantly in a short period of time.

These types of databases have unique characteristics. Firstly, the tables look a lot like a log file to which new rows are appended. New rows can be inserted but never updated, except when such rows must be deleted because they are too old for the business. Moreover, the new rows are always inserted in their natural order (from least to most recent) except for some rare exceptions, for example when the gateways are off-line. In these cases the data will be retransmitted in bulk, but still in chronological order.

6.1 Data types and Indexes consideration

The table in which the information from the gateways is stored has been deliberately kept very simple. The columns present information on:

- timestamp
- gateway identifier
- identifier of the user who generated it (optional)
- event type
- JSON payload

To speed up data entry operations, no foreign keys to keys in other tables have been added. Since the data types are very small, a 2-byte integer is largely sufficient. Since the data can have a heterogeneous structure, a Jsonb field was used to save the payload.

```
CREATE TABLE "public"."gateways_data" (
    "logged_at" timestamp NOT NULL DEFAULT now(),
    "gateway_id" uuid NOT NULL,
    "user_id" uuid,
    "type" int2,
    "payload" jsonb,
    PRIMARY KEY ("logged_at","gateway_id")
);
```

To allow fast access to the rows of the table, it is necessary to prepare the right indexes. The default index used by PostgreSQL is a BTree index. This type of index is fine for different types of data, but as the data grows it tends to be intensive to maintain and grow in size. BTree indexes are particularly good when you want to derive a single tuple from the table. For time series data, however, the most common scenario involves querying over a range of time. In this specific case it is much more useful to use a BRIN (Block Range Index): instead of keeping track of every single time value, a **BRIN** index keeps track of the minimum and maximum value over a range of pages in the table. Since in the time series database there is a direct correlation between time value and physical page on the disk, this type of index is particularly efficient. [10]

```
CREATE INDEX gateways_data_logged_at
    ON gateways_data
 USING BRIN (logged_at)
     WITH (pages_per_range = 64)
```

The higher the pages per range, the lower the selectivity of our index, so the value of pages_per_range must be weighted according to the queries that are typically made. Since the gateway_id column uses UUID, an index of type BTREE has been chosen because it is the one that works best with this type of data.

```
CREATE INDEX gateways_data_gateway_id
    ON gateways_data
 USING BTREE (gateway_id, logged_at DESC)
```

No matter how well you optimize indexes and data types, there will come a time when performance will begin to degrade because the overhead introduced by indexes is no longer negligible. The natural solution is to keep datasets and their respective indexes small.

6.2 Table partitioning as solution

For databases with extremely large tables, partitioning is the ideal solution to keep performance consistent over time and make maintenance much easier. Partitioning splits a table into multiple tables, and generally applications accessing the table don't notice any difference, other than being faster to access the data that it needs. Splitting the table into multiple tables allows execution of the queries over much smaller tables and indexes to find the data needed. Even without any particular optimization, scanning a 10GB table will definitely be faster than scanning a 100GB table. The advantages are also present in the management of indexes: each table will have a smaller dataset and consequently smaller indexes that are easier to update. If data needs to be periodically deleted, purging also becomes faster: delete statements will create dead rows that need to be vacuumed. With table partitioning a multiple delete becomes a simpler and faster table drop.

6.3 PostgreSQL partitioning methods

Postgres provides three built-in partitioning methods:

- List Partitioning: partition a table by a list of known values, like categories or countries
- Hash Partitioning: partition a table by applying a hash function on the partition key. This method is useful when there is no list of predefined values on which to perform a logical division of the data, such as product ID codes.
- Range Partitioning: partition a table by a range of values. The table containing data can be divided by year, month or week. This naturally fits the needs of time series databases.

The choice of partitioning criteria is a crucial aspect that can positively or negatively affect performance. The choice must respect the nature of the data but also how they are accessed.

In our specific case the data is read on a weekly basis and in chronological order. Using range partitioning a split of the tables by date has been applied, creating a partition for each week of the year.

We are going to recreate a new table, specifying partitions by range of the field "logged$_a t$J.

```
CREATE TABLE "gateways_data" (
"logged_at" timestamp NOT NULL DEFAULT now(),
"gateway_id" uuid NOT NULL,
"user_id" uuid,
"type" int2,
"payload" jsonb,
PRIMARY KEY ("logged_at", "gateway_id")
)
PARTITION BY RANGE ("logged_at");
```

Now we have to create tables for each weeks we want to keep on our database:

```
CREATE TABLE "gateways_data_2021w13" PARTITION OF "gateways_data"
FOR VALUES FROM ('2021-03-29') TO ('2021-04-04');

CREATE TABLE "gateways_data_2021w14" PARTITION OF "gateways_data"
FOR VALUES FROM ('2021-04-05') TO ('2021-04-11');
```

Inserting data will not require any particular addition to our query. Let's create some rows with the following query:

```
INSERT INTO "gateways_data"
("logged_at", "gateway_id", "user_id", "type", "payload")
VALUES
('2021-04-03 9:00:01', '...', null, 1, '{ "foo": "bar"}'),
('2021-04-04 9:00:00', '...', null, 1, '{ "foo": "bar"}'),
('2021-04-05 9:00:00', '...', null, 1, '{ "foo": "bar"}'),
('2021-04-06 9:00:00', '...', null, 1, '{ "foo": "bar"}'),
('2021-04-07 9:00:00', '...', null, 1, '{ "foo": "bar"}'),
('2021-04-08 9:00:00', '...', null, 1, '{ "foo": "bar"}'),
('2021-04-09 9:00:00', '...', null, 1, '{ "foo": "bar"}'),
('2021-04-10 9:00:00', '...', null, 1, '{ "foo": "bar"}'),
('2021-04-11 9:00:00', '...', null, 1, '{ "foo": "bar"}');
```

If we do a SELECT query over the main table, it will returns all rows as expected:

```
SELECT
"logged_at", "type", "payload"
FROM "gateways_data";

| logged_at           | type | payload         |
|---------------------|------|-----------------|
| 2021-04-03 09:00:01 |    1 | {"foo": "bar"} |
| 2021-04-04 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-05 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-06 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-07 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-08 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-09 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-10 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-11 09:00:00 |    1 | {"foo": "bar"} |
```

Exploring the newly created partitioned tables instead, we can see that 2 of the 9 rows inserted have been created in the first table, while the remaining 7 in the second one. The DBMS took care of everything by making the partitioning operation transparent to the user.

```
SELECT
"logged_at", "type", "payload"
FROM
"gateways_data_p2021w13";

| logged_at           | type | payload        |
|---------------------|------|----------------|
| 2021-04-03 09:00:01 |    1 | {"foo": "bar"} |
| 2021-04-04 09:00:00 |    1 | {"foo": "bar"} |


SELECT
"logged_at", "type", "payload"
FROM
"gateways_data_p2021w14";

| logged_at           | type | payload        |
|---------------------|------|----------------|
| 2021-04-05 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-06 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-07 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-08 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-09 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-10 09:00:00 |    1 | {"foo": "bar"} |
| 2021-04-11 09:00:00 |    1 | {"foo": "bar"} |
```

Despite the simplicity of the commands, creating individual table partitions is a tedious and repetitive process. We could manage this at the application level, however the pg_partman extension allows us to manage this in an automated manner [25].

This is simple as running this query:

```
CREATE EXTENSION pg_partman;
SELECT
partman.create_parent ('public.gateways_data',
'logged_at',
'native',
'weekly');
```

With this query pg_partman will periodically create new partition table on weekly base and if need will prune tables containing old data, keeping the whole system efficient during time.

```
UPDATE
partman.part_config
SET
retention_keep_table = FALSE,
retention = '3 month'
WHERE
parent_table = 'public.gateways_data';
```

Chapter 7

Adoption of DevOps practices

Software development cycles have become much tighter in recent years. With the rise of Agile methodologies, developers are trying to make new features available as quickly as possible: releases that used to be monthly can now be daily or even hourly. To achieve this, different developers can work in parallel on different features or modules of the same application and individually commit their changes to a shared repository. Therefore, a system that facilitates all testing and integration operations is vital. It is in this scenario that DevOps practices have spread and are becoming the de facto standard of recent years.

7.1 The Agile Development

The Agile model, as opposed to the Waterfall model or other traditional software processes, it proposes an approach less structured and focalized on the objective to deliver to the customer, quickly and frequently, working and quality software.

In the Waterfall model the phases of Analysis, Design, Implementation and Testing follow each other in a chronological and well-defined sequence: a new phase starts when the previous one ends. A change of requirements, or the integration of new functional specifications lead to the need to rework some components of the software, causing inefficiency and waste of time.

The Agile model requires these phases to be repeated regularly, as the development process is divided into individual units that are analyzed, implemented and tested independently. A crucial aspect in the Agile model is the acquisition of feedback: the sooner errors in realization or specification are identified, the sooner they can be corrected, decreasing the related costs.

Such a work organization also allows for high parallelization, as each individual unit can be worked on by a different team at the same time. The high efficiency that can be achieved, however, must also be supported by tools that facilitate all routine activities such as testing, distribution and production. This is where the concept of **DevOps** comes from, focusing on the concepts of automation and timeliness.

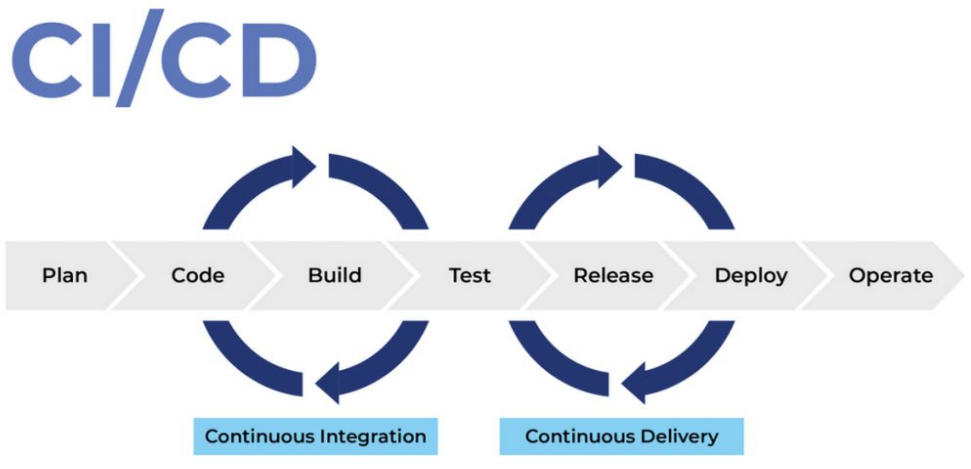


Figure 7.1: CI/CD cycle

7.2 Continuous Integration and Deployment

Continuous Integration is about automating the build phases through pipelines: developers regularly add changes to code in a centralized repository, with builds and tests performed automatically in order to find and fix bugs earlier, improve software quality, and reduce the time required to validate and publish new updates. When developers commit changes to their code (using Git for example), the build management system automatically creates a build and tests it. If the test fails, the system notifies the team to fix the code. This practice helps software teams and maintains code in a deployable state.

7.3 Continuous Delivery and Deployment

Every time the pipeline builds and tests the code changes, the **Continuous Delivery automatically prepares new artifacts to be deployed**. Continuous Delivery is the natural consequence of Continuous Integration, as it allows all code changes to be distributed to the testing and/or production environment after the build phase, allowing developers to automate testing and verify the application of updates before making them available to customers. These tests can include interface testing, load testing, integration testing, API reliability testing, and more.

Continuous Deployment completes **Continuous Delivery**: new artifacts are automatically released to various environments without explicit approval or manual steps. The new deployment is monitored with health-checks: in case of problems rollback to the previous version can be done.

7.4 Benefits of CI/CD

The benefits of adopting a CI/CD system are obvious:

- **Automated software release** (across multiple environments): continuous deployment allows developers to prepare builds and test code changes for production release more efficiently and quickly.
- **Increased developers productivity**: Freeing developers from manual and time-consuming tasks improves productivity, as well as incentivizing the adoption of practices that reduce the number of errors and bugs in software (testing).
- **Faster detection and resolution of bugs**: more comprehensive and frequent testing allows bugs to be detected before they become serious problems.
- **Faster updates**: new versions are released faster and more frequently because builds are always available, tested, and ready for deployment.

7.5 The selected tools

Declining Devops principles requires architecting a CI/CD pipeline that relies on a set of tools that may also be open-source that works together with each other.

- **Source Code repository**: *Git*
  is an archive in which the various modifications of the source code are saved and versioned. It allows you to keep track of all the changes made over time, but most importantly it allows multiple developers to work on the same codebase at the same time. Git was born in 2005 from the hands of Linus Torvalds and has become the most popular and used system in the world. It is the basis of popular services such as Github, BitBucket, GitLab.
- **Artifacts Format and Registry**: *Docker Docker Registry*
  Each new commit corresponds to an artifact build. In our case, a new Docker image is produced and saved to a local registry.

- **Orchestration**: *Docker Swarm*
  Docker swarm is a container orchestration tool that allows managing multiple containers deployed across multiple host machines. One of the key benefits associated with the operation of a docker swarm is the high level of availability offered for applications.

- **CI/CD pipeline**: *Drone.io*
  Drone is a Continuous Delivery system built on container technology. It allows defining and executing Pipelines inside Docker containers writing them using a simple YAML configuration file.

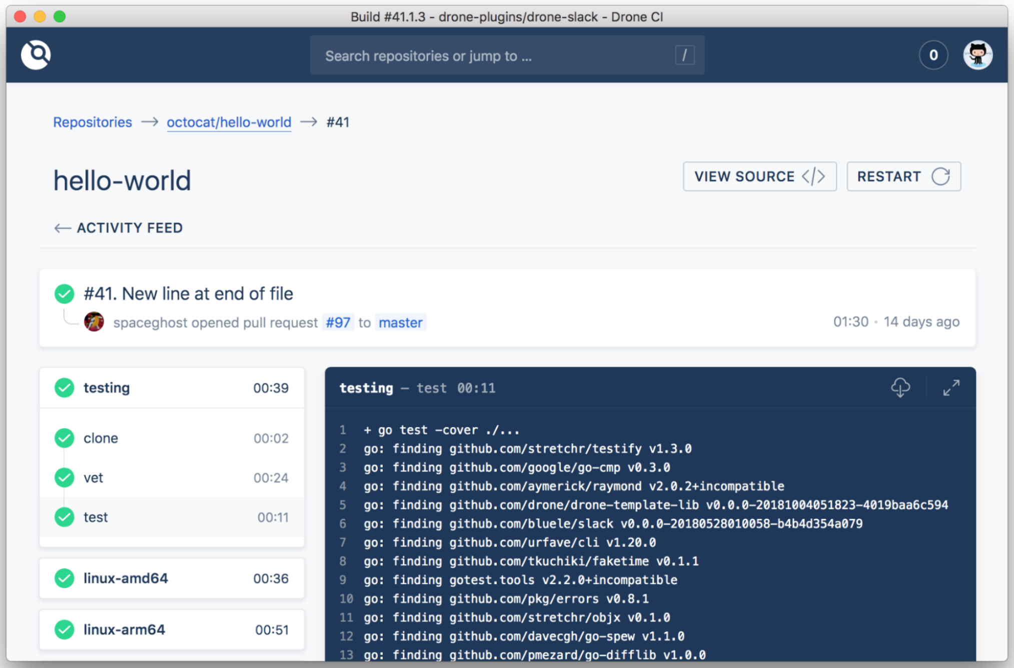
7.6 Brief Comparison of CI/CD tools

Although there are several CI/CD solutions on the market, the choice has been restricted to all those solutions that were open-source and installable on-promise. The simplicity of installation and configuration are a decisive plus for the choice.

- **Jenkins**: is perhaps the most mature and well-known project. It is an open-source Java-based automation server and has a solid community and has 1400+ plugins for every need. It's a highly versatile and powerful tool, however this is reflected in higher effort given the complexity of configurations and maintenance. It does not support Docker natively, but only through plugins. Scaling is possible by installing multiple runners on multiple hosts.

- **GitLab CI**: This tool allows you to host on your servers all the tools needed for the entire software lifecycle without the need of plugins: code repository, Docker repository, pipeline management. Unfortunately, these features have proven to be redundant with the existing infrastructure.

- **Drone.io**: despite being the youngest project, it started as a container-native and is written in Go. It integrates with the versioning solution of your choice (Github, Bitbucket, etc...) through webhooks and thanks to the use of Docker can be installed virtually anywhere.
  Docker is widely used in pipeline execution too: each step is executed in a dedicated container that can be customized according to your needs. It is therefore possible to take advantage of knowledge already gained without learning a new scripting language. Official plugins extend the functionality, and allow you to compose complex workflows with a block logic: clone, build, test, deploy.


Build #41.1.3 - drone-plugins/drone-slack - Drone CI
Search repositories or jump to ...
Repositories → octocat/hello-world → #41
hello-world
VIEW SOURCE
RESTART
← ACTIVITY FEED
#41. New line at end of file
spaceghost opened pull request #97 to master
01:30 · 14 days ago
testing 00:39
clone 00:02
vet 00:24
test 00:11
linux-amd64 00:36
linux-arm64 00:51

testing – test 00:11

```
+ go test -cover ./...
go: finding github.com/stretchr/testify v1.3.0
go: finding github.com/google/go-cmp v0.3.0
go: finding github.com/aymerick/raymond v2.0.2+incompatible
go: finding github.com/drone/drone-template-lib v0.0.0-20181004051823-4019baa6c594
go: finding github.com/bluele/slack v0.0.0-20180528010058-b4b4d354a079
go: finding github.com/urfave/cli v1.20.0
go: finding github.com/tkuchiki/faketime v0.1.1
go: finding gotest.tools v2.2.0+incompatible
go: finding github.com/pkg/errors v0.8.1
go: finding github.com/stretchr/objx v0.1.0
go: finding github.com/davecgh/go-spew v1.1.0
go: finding github.com/pmezard/go-difflib v1.0.0
```

Figure 7.2: Drone.io task execution view

7.7 Defining a pipeline

Pipelines help to automate steps in your software delivery process, such as initiating code builds, running automated tests, and deploying to a staging or production environment. Unlike other solutions, where the definition of a pipeline can be a long and complex operation, Drone provides that the pipeline is defined in a YAML file saved within the Git repository.

7.7.1 Structure of a pipeline

Listing 7.1: Drone.io Pipeline structure

```yaml
kind: pipeline
name: Building Backend

steps:
# installing required node dependencies
- name: Install Dependencies
  image: node
  commands:
  - yarn install
# building sources
- name: Building Sources
  image: node
  commands:
  - yarn build
# running migrations and seed file
- name: Building Sources
  image: node
  commands:
  - yarn migrate
  - yarn seed
# running unit tests
- name: Running Tests
  image: node
  commands:
  - yarn test
# creating new docker image and save to registry
- name: Creating Image
  image: plugins/docker
  settings:
    username: foo
    password: bar
    repo: project/module
    tags: latest
```

As we can see, each step of the pipeline is defined by a set of attributes (name, image and commands) and will be executed if the previous one has been completed successfully. The Docker image used can be an official image (available on Docker HUB) or a custom image made for the purpose.

The pipeline will be executed on each commit made to the repository. Drone gives you the possibility to set triggers, such as a push to a specific branch (Listing 7.2), or following a pull request (Listing 7.3).

Listing 7.2: Pipeline example n.1

```
kind: pipeline
name: Building Backend

steps:
  - ....
trigger:
  branch:
  - master
  - feature/*
```

Listing 7.3: Pipeline example n.2

```
kind: pipeline
name: Building Backend

steps:
  - ....
trigger:
  branch:
  - master
  event:
    include:
    - push
    - pull_request
```

7.7.2 Running Services

In order to perform integration tests, it may be necessary to have services available such as a DBMS or shared storage such as Redis. (Listing 7.4) These containers will be initiated and made available to the other steps in the pipeline. This operation allows you to replicate for each feature you are developing a faithful reproduction of an environment that is fully corresponding to the production one. Tests executed will be representative of the situation that could occur in production, allowing to easily intercept problems and verify the resolution thanks to the repeatability of tests at any time.

Listing 7.4: Pipeline Services

```
kind: pipeline
type: docker
name: default

services:
- name: redis-server
  image: redis
- name: postgres-db
  image: postgres:12.2-alpine
steps:
  - ....
```

7.7.3 Executing Deployment

As soon as an artifact has been generated and passed all quality tests, it can be deployed on a development or production server. This phase is the so-called **continuous deployment**. In the Drone ecosystem we find plugins that can take care of releasing the new code on known cloud environments such as Amazon AWS, Google Cloud or Kubernetes. If these plugins are not suitable, it is possible to run commands directly on remote machines using SSH. It is not the most fashionable solution but it allows you to shape the deployment process according to your needs.

Listing 7.5: Pipeline Deployment

```
kind: pipeline
type: docker
name: default

services:
- name: redis-server
  image: redis
- name: postgres-db
  image: postgres:12.2-alpine
steps:
  - ....

- name: ssh commands
  image: appleboy/drone-ssh
  settings:
    host: 1.2.3.4
    username:
        from_secret: ssh_username
    key:
        from_secret: ssh_key
    script:
      - echo hello
      - echo world
```

7.7.4 Sending Notification

Once the pipeline has completed its course, or errors have occurred, all that remains is to notify the appropriate team. Notifications can be sent via email, Telegram, or even on a Slack channel. Like everything seen so far, notifications consist of a step executed by a Docker image.

Chapter 8

Conclusions

The development of this thesis was an opportunity for a multidisciplinary study focusing on the world of the Internet of Things. The project was followed in all its phases: from the realisation of the first prototypes, to the definition of the system specifications in agreement with the client, and the concrete development of the functional components described in the previous chapters.

Although JavaScript was the fundamental language used, there were several design challenges to which a solution had to be found.

The backend cloud component is the largest and also the most critical, as it represents the core element of the platform. It was essential to structure it in such a way as to make everything maintainable and above all evolutionary.

Software design principles such as the Separation of Responsibilities (and all the 5 SOLID principles that follow from it) combined with the main patterns, have made the code base organised, reusable and above all comprehensible even to those approaching the project for the first time.

In order to make the platform scalable according to the volume of traffic to be managed, it was made decentralised with a microservices architecture. One benefit of this is improved failure tolerance, thanks to the elimination of single points of failure. However, such an architecture requires effective communication. The ideal solution in this scenario is the use of queues and the *old-school* Producer and Consumer pattern.

NodeJS and its asynchronous nature performed exceptionally well, but tools such as Redis, MQTT and Docker were key allies in achieving this result. The use of Docker in particular has facilitated all phases of the implementation process: with just a few steps, it is possible to configure a development environment that is completely similar to the production environment and consistent across all team members. The same configurations are then used within the CI/CD pipeline for testing the artifacts produced and their automatic deployment to the staging and production machines.

The software on the gateway, on the other hand, imposed totally different needs. Computational and memory resources are significantly lower than those of any server, so everything must be used carefully to avoid memory leaks that could lead to deadlocks. The software must also take into account the possible lack of connectivity and take mitigating action to ensure that the system will still operate correctly. For interfacing with sensors and actuators, no high-level libraries were available, so everything was implemented by reading and writing from file descriptors or worse, serial interfaces. Readings have to be normalised and cleaned of glitches using debouncing before being saved or transmitted.

The bridge between the gateway and mobile app was instead the BLE protocol, in which the former operates as a peripheral and the latter as the central. This required structuring the information schematically and concisely for transmission without requiring everything to be split into smaller fragments that remain within the 20-byte payload size limits. Last but not least, optimising the database is a guarantee that performance remains constant over time.

Despite the fact that only one programming language was used, the solutions to the design problems were all different and allowed transversal skills to be developed and strengthened, thus consolidating the professional profile of the full-stack developer.

8.1 Future works and improvements

One of the explicit requirements of the project is its expandability. The availability of environmental data can be important in the creation of a smart city, and gateways installed throughout the territory can be probes capable of collecting information on the surrounding environment. Future developments will undoubtedly involve the integration of new sensors to extend the potential of the platform.

The platform is currently set up to collect information from sensors of various kinds, such as those for particulate matter and electromagnetic fields. If activated on a large scale, it is easy to assume a huge amount of data to be analysed, so a Big Data Analysis engine becomes a fundamental aid to the correlation and analysis of data, fluctuations in time and variations on a geographical basis. Two of the most famous tools are Apache Hadoop and Apache Spark, to name just a few.

From an architectural point of view, porting to Kubernetes orchestration technology could be of strategic value, making it possible to benefit from the managed services of major cloud service providers such as Google or AWS.

Bibliography

[1] *Internet of things*. en. Page Version ID: 1041593457. Aug. 2021. URL: https://en.wikipedia.org/w/index.php?title=Internet_of_things&oldid=1041593457 (cit. on p. 1).

[2] Node.js. *Node.js*. en. URL: https://nodejs.org/en/ (cit. on p. 8).

[3] *V8 JavaScript engine*. URL: https://v8.dev/ (cit. on p. 8).

[4] *Design overview — libuv documentation*. URL: http://docs.libuv.org/en/v1.x/design.html (cit. on p. 8).

[5] *About npm | npm Docs*. URL: https://docs.npmjs.com/about-npm (cit. on p. 8).

[6] *Yarn*. en. URL: https://yarnpkg.com/en/ (cit. on p. 8).

[7] Dhruv Patel. *Demystifying Vue.js internals*. en. May 2018. URL: https://medium.com/js-imaginea/the-vue-js-internals-7b76f76813e3 (cit. on p. 9).

[8] *Apache Cordova*. URL: https://cordova.apache.org/ (cit. on p. 9).

[9] *Architectural overview of Cordova platform - Apache Cordova*. URL: https://cordova.apache.org/docs/en/latest/guide/overview/index.html (cit. on p. 9).

[10] *CREATE INDEX*. en. Aug. 2021. URL: https://www.postgresql.org/docs/12/sql-createindex.html (cit. on pp. 10, 41).

[11] *NOTIFY*. en. Aug. 2021. URL: https://www.postgresql.org/docs/12/sql-notify.html (cit. on p. 10).

[12] *Redis*. URL: https://redis.io/ (cit. on p. 10).

[13] *MQTT - The Standard for IoT Messaging*. URL: https://mqtt.org/ (cit. on p. 11).

[14] *An Open-Source, Cloud-Native, Distributed MQTT Broker | EMQ X*. URL: https://www.emqx.io/ (cit. on p. 12).

[15] *Clean Coder Blog*. URL: `https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html` (visited on 03/14/2022) (cit. on p. 13).

[16] Sam Newman. *Building microservices: designing fine-grained systems*. First Edition. OCLC: ocn881657228. Beijing Sebastopol, CA: O'Reilly Media, 2015. ISBN: 978-1-4919-5035-7 (cit. on p. 15).

[17] Michael Szczepanik. *Backend for frontend (BFF) pattern— why do you need to know it?* en. Nov. 2021. URL: `https://medium.com/mobilepeople/backend-for-frontend-pattern-why-you-need-to-know-it-46f94ce420b0` (visited on 03/14/2022) (cit. on p. 17).

[18] *Sam Newman - Backends For Frontends*. URL: `https://samnewman.io/patterns/architectural/bff/` (visited on 03/14/2022) (cit. on p. 17).

[19] Sergio Prado. *Linux kernel GPIO user space interface*. en. URL: `https://embeddedbits.org/new-linux-kernel-gpio-user-space-interface/` (cit. on p. 21).

[20] *GATT (Services and Characteristics) - Getting Started with Bluetooth Low Energy [Book]*. en. ISBN: 9781491949511. URL: `https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html` (cit. on p. 24).

[21] Alexis Duque. *Deep Dive into Bluetooth LE Security*. en. Mar. 2018. URL: `https://medium.com/rtone-iot-security/deep-dive-into-bluetooth-le-security-d2301d640bfc` (cit. on p. 27).

[22] *Understanding Bluetooth Security*. en. URL: `https://duo.com/decipher/understanding-bluetooth-security` (cit. on p. 27).

[23] *MessagePack: It's like JSON. but fast and small.* URL: `https://msgpack.org/` (cit. on p. 29).

[24] *Visual Studio App Center | iOS, Android, Xamarin & React Native*. URL: `https://appcenter.ms/` (cit. on p. 38).

[25] *PG Partition Manager*. original-date: 2012-09-05T05:04:46Z. Aug. 2021. URL: `https://github.com/pgpartman/pg_partman` (cit. on p. 45).