Projeto

Mestrado em Engenharia Informática – Computação Móvel

Domatica All Connect Core – Broker IoT Modular

Rúben Daniel Dos Santos e Sousa Ferreira

Leiria, setembro de 2017

Projeto

Mestrado em Engenharia Informática – Computação Móvel

Domatica All Connect Core – Broker IoT Modular

Rúben Daniel Dos Santos e Sousa Ferreira

Projeto de Mestrado realizada sob a orientação do Doutor João da Silva Pereira, Professor da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria.

Leiria, setembro de *2017*

Esta página foi intencionalmente deixada em branco

Agradecimentos

A conclusão deste projeto não serial possível sem a ajuda de algumas pessoas.

Aproveito assim este espaço para prestar o devido agradecimento.

Em primeiro lugar, gostaria de agradecer à entidade de acolhimento – Domatica Global Solutions – e aos seus colaboradores, por todo o apoio e condições de trabalho que me foram oferecidas, bem como por toda a hospitalidade. Gostaria também de destacar algumas pessoas que de forma mais direta tiveram um papel imprescindível no desenvolvimento deste projeto.

Ao Engenheiro Samuel Silva, CEO da Domatica, pela incrível disponibilidade demonstrada, mesmo quando o seu tempo era bastante limitado, pela orientação, pelas várias horas que se disponibilizou para me ajudar no desenvolvimento, pela sua preocupação e por me ter tornado melhor profissional e mais capacitado. Sem dúvida alguma que sem ele o resultado final não era o mesmo.

Ao Pedro Pina, diretor e vice-presidente da Domatica, por todo o seu apoio e orientação durante a fase de desenvolvimento.

Ao Carlos Reis, por todo o apoio logístico fornecido no escritório em Leiria, pela integração na empresa e também por toda a ajuda fornecida durante o desenvolvimento do projeto.

Por outro lado, gostaria de agradecer ao orientador, o professor Doutor João Pereira, por toda a orientação e apoio dado durante o desenvolvimento do projeto e do relatório.

À Escola Superior de Tecnologia e Gestão (ESTG) do Instituto Politécnico de Leiria, o meu muito obrigado uma vez mais pelas excelentes condições oferecidas, bem como a todos os professores do Mestrado, que de uma forma ou outra, contribuíram para os conhecimentos adquiridos.

Em último lugar e não menos importante, queria agradecer aos meus familiares por todo o apoio, bem como pelas condições que me proporcionaram uma vez mais, para chegar ao fim de mais um objetivo.

Esta página foi intencionalmente deixada em branco

Resumo

Com a constante evolução tecnológica dos últimos anos, os dispositivos inteligentes vieram uma vez mais mudar a forma como interagimos no nosso quotidiano. O termo *Internet Of Things* (Internet das Coisas) é o conceito dado que define a revolução tecnologia de conferir inteligência a objetos do nosso dia-a-dia, ao permitir a sua comunicação com diversos tipos de dispositivos inteligentes com ligação à internet. Esta revolução encontra-se atualmente em crescimento, sendo cada vez mais as empresas a apostar neste conceito. Esta aposta por parte das empresas acaba por trazer alguns desafios. O principal objetivo é a atual falta de recursos por parte das empresas, seja em termos de dinheiro ou termos de pessoas. Deste modo existe uma maior necessidade de otimizar esses recursos de forma mais eficiente, sendo a *Internet Of Things* o principal motor para essa otimização. Em grande parte das empresas, a monitorização dos seus recursos já é aplicada, no entanto a mesma nem sempre chega de forma simples e rápida a quem tem de tomar as devidas decisões no seio da empresa, como por exemplo o departamento financeiro. Deste modo, as novas soluções inteligentes, permitem que de uma forma rápida, automática e integrada, sejam recolhidos todos os dados, para que cheguem em tempo real aonde devem chegar. No entanto, estas soluções não passam apenas por monitorização, mas por controlo, sendo possível controlar à distância um conjunto de funcionalidades, deste controlo de iluminação até ao controlo de máquinas industriais de produção. Assim os utilizadores deste tipo de soluções, sentem a necessidade de integração dos dispositivos inteligentes, de forma também ela simples e rápida, no seio das suas necessidades. Existe a necessidade de desenvolvimento de plataformas que permitam a comunicação entre os diversos objetos, com estes dispositivos inteligentes, para que seja possível a troca de dados e controlo entre ambos. Com este tipo de plataformas, o nível de abstração e complexidade na integração de soluções *Internet Of Things*, vai diminuir drasticamente permitindo a otimização de uma coletânea de recursos e consecutivamente contribuir para uma maior eficiência das pessoas e das empresas.

Este projeto teve como finalidade a criação de uma API que permitisse a comunicação com a *cloud* e consequentemente com os dispositivos IoT facilitando assim a integração e desenvolvimento de soluções IoT.

Palavras-chave: revolução, Internet of Things, inteligência, eficiência, monitorização

Esta página foi intencionalmente deixada em branco

Abstract

With the constant technological evolution of recent years, intelligent devices have come once again to change the way we interact in our daily lives. The term Internet of Things is the given concept that defines the technology revolution of conferring intelligence to objects of our daily life, allowing its communication with several types of intelligent devices with internet connection. This revolution is currently growing, with more and more companies betting on this concept. This bet on the part of the companies ends up bringing some challenges. The main objective is the current lack of resources by companies, either in terms of money or terms of people. This way there is a greater need to optimize these resources more efficiently, with Internet Of Things being the main engine for this optimization. In most companies, the monitoring of their resources is already applied, however it does not always arrive in a simple and quick way to those who should make the right decisions within the company, such as the finance department. In this way, the new intelligent solutions allow a fast, automatic and integrated way, to collect all the data, so that they arrive in real time where they should arrive. However, these solutions are not only monitored, but controlled, and it is possible to remotely control a set of functionalities from this lighting control to the control of industrial production machines. So, the users of this type of solutions, feel the need to integrate smart devices, also simple and fast, within their needs. There is a need to develop platforms that allow communication between the various objects, with these intelligent devices, so that it is possible to exchange data and control between both. With this type of platforms, the level of abstraction and complexity in the integration of Internet Of Things solutions will drastically decrease, allowing the optimization of a collection of resources and consecutively contributing to greater efficiency of people and companies.

This project aimed to create an API that would allow communication with the cloud and consequently with the IoT devices, thus facilitating the integration and development of IoT solutions.

Keywords: revolution, Internet of Things, intelligence, efficiency, monitoring

Esta página foi intencionalmente deixada em branco

Lista de figuras

Figura 1 - Arquitetura Azure IoT Hub 9
Figura 2 - RoboMQ 10
Figura 3 - REST API (DeviceHive) 11
Figura 4 - Comunicação 12
Figura 5 - Fluxos de código (Node-RED) 14
Figura 6 - Topologia Zetta 16
Figura 7 - Conceitos Thingworx 17
Figura 8 - Conectores ThingWorx 18
Figura 9 - IoT Challenge 19
Figura 10 - Metodologia Scrum 22
Figura 11 - Domatica Quantum Node 26
Figura 12 - Quantum Node Extender 27
Figura 13 - Domatica Quantum Cloud 29
Figura 14 - REST API 30
Figura 15 - Message Brocker 31
Figura 16 - Arquitectura AMQP 33
Figura 17 - Arquitetura MQTT 34
Figura 18 - MQTT vs CoAP 36
Figura 19 - RabbitMQ 37
Figura 20 - Class Person 43
Figura 21 - Formato JSON 43
Figura 22 - HTTP vs HTTPS 71
Figura 23 - Man-in-the-middle 71
Figura 24 - GET vs POST 72

Esta página foi intencionalmente deixada em branco

Lista de siglas

IoT – Internet of Things

MIT - Massachusetts Institute of Technology

DQS – Domatica Quantum System

DQN – Domatica Quantum Node

DQC – Domatica Quantum Cloud

API – Application Programming Interface

REST – Representational State Transfer

WEB – World Wide Web

HTTP – Hypertext Transfer Protocol

XML - eXtensible Markup Language

JSON – JavaScript Object Notacion

AMQP - Advanced Message Queuing Protocol

MQTT - Message Queuing Telemetry Transport

CoAP - Constrained Application Protocol

SDK - Software Development Kit

URL - Uniform Resource Locator

HTTPS - Hyper Text Transfer Protocol Secure

HTTP - Hyper Text Transfer Protocol

SSL – Secure Socket Layer

TLS – Transport Layer Secutiry

M2M – Machine-to-Machine

IP - Internet Protocol

Esta página foi intencionalmente deixada em branco

Índice

AGRADECIMENTOS III

RESUMO V

ABSTRACT VII

LISTA DE FIGURAS IX

LISTA DE SIGLAS XI

ÍNDICE XIII

1. INTRODUÇÃO 1

1.2 Entidade de acolhimento 2

1.3 Motivação e objetivos do projeto 3

1.4 Estrutura do relatório 5

2. ESTADO DA ARTE 5

2.1 Web Services 6

2.1.2 Representações Web Service 6

2.2 Aplicações e Serviços 8

2.2.1 Azure IoT Hub 8

2.2.2. RoboMQ 9

2.2.3 DeviceHive 10

2.2.4 macchina.io 11

2.2.5 Mainspring 12

2.2.6 Node-RED 13

2.2.7 Open Connectivity Foundation 14

2.2.8 Zetta 15

2.2.9 ThingWorx 16

2.2.10 IoT real time data acquisition using MQTT protocol 19

2.2.11 Sense Tecnic Systems 20

2.3 Advanced Message Queuing Protocol (AMQP) 20

3. METODOLOGIA E GESTÃO DE PROJETO 21

3.1 Metodologia 21

3.2 Gestão de Projeto 23

3.3 Controlo de versões 24

4. ARQUITETURA 25

4.1 Domatica Quantum Nodes (DQN) 25

4.2 Domatica Quantum Cloud (DQC) 28

4.2.1 Comunicação 29

4.2.2 Protocolo de Comunicação 31

4.2.3 Camada L0 38

4.2.4 Camada L1 41

5. IMPLEMENTAÇÃO, TESTES E SEGURANÇA 44

5.1 Funcionalidades Implementadas 44

5.1.1 Métodos de autenticação 44

5.1.2 Gestão de utilizadores 46

5.1.3 Gestão das Entidades 49

5.1.4 Gestão das Gateways/Nodes IoT 50

5.1.5 Gestão de Tokens 63

5.1.6 Armazenamento dos dados 65

5.2 Testes e Ferramentas 66

5.2.1 Testes Unitários 67

Camada L0 67

Camada L1 67

5.2.2 Testes de aceitação 68

Resultados 68

5.3 Segurança 69

5.3.1 Ligação HTTPS 70

5.3.2 Core API 72
5.3.3 Base de dados 73

6. CONCLUSÃO **76**

6.1 Trabalho Futuro **77**

BIBLIOGRAFIA **79**

ANEXOS **84**

1. Introdução

Internet of things (IoT), conhecida também como a internet das coisas, é a capacidade de interligar à internet um vasto número de objetos, permitindo que estes comuniquem entre si, ficando mais eficientes e com novas funcionalidades [1].

Nos últimos tempos o conceito de *Internet of things* tem tido um crescimento exponencial, traduzindo-se num impacto quer no modo em que vivemos, mas também como trabalhamos [2].

Segundo Kevin Ashton, do Instituto de Tecnologia de Massachussets (MIT) e fundador do termo "Internet das coisas", a falta de tempo na rotina quotidiana das pessoas irá fazer com que estas sintam a necessidade de se conectar à internet de diversas maneiras para que assim se sintam auxiliadas na realização das suas tarefas [3]. Estes dispositivos inteligentes poderão atuar em diversos sectores, como é exemplo os Hospitais e Clínicas, Agricultura, *Smart Cities*, e diversas áreas industriais e de manufaturação.

Por exemplo, no seio dos Hospitais e Clínicas, os pacientes estão muitas vezes ligados a dispositivos que medem os batimentos cardíacos. Estes dados podem ser recolhidos e enviados em real time para um sistema que faça o controlo destes dados.

Por outro lado, no mundo da Agropecuária é provavelmente onde a IoT mais tendo atuado e crescido. Atualmente existem já diversas plantações com diversos sensores espalhados, podendo estes dar informações ao nível da temperatura, umidade do solo, probabilidade de ocorrência de precipitação, velocidade do vento, entre outras [1]. Por outro lado, também os animais podem ser conectados com certos tipos de sensores, como é o caso de um chip, que faça recolha de informação, plano de vacinação, rastreamento, etc., para ser sempre possível consultar o seu histórico e fazer o devido acompanhamento [1]. Estes são apenas breves referências a uma lista de vários serviços e sectores onde a IoT pode atuar, sendo a Domatica Global Solutions uma empresa desenvolvedora deste tipo de soluções, tanto a nível de *hardware* como *software*.

Atualmente a Domatica Global Solutions já tem em funcionamento os seus próprios dispositivos inteligentes – Domatica Quantum Nodes (DQN) – que por sua vez estão interligados à *cloud* – Domatica Quantum Cloud (DQC). Estes dipositivos inteligentes permitem executar um conjunto de operações, desde a leitura de informação de diversos sensores (temperatura, umidade, CO2, entre outros), bem como atuar sobre outros

dispositivos, como por exemplo um led, permitindo alterar o estado deste (ligado/desligado).

Como já foi referido, hoje em dia, a *IoT* passa pela relação entre o mundo físico e o mundo digital. Na comunicação entre estes dois ambientes o conceito de *Cloud Computing* apresenta um papel primordial. A mesma irá disponibilizar capacidade computacional flexível o suficiente para armazenar a grande quantidade de recursos provenientes dessa comunicação. Além do armazenamento de todos esses recursos, permite ainda que os mesmos sejam acedidos, monitorizados e controlados à distância graças à facilidade que atualmente se navega na internet e acede a esses recursos disponibilizados pela *Cloud*.

O propósito deste projeto passou pelo desenvolvimento de uma *IoT* Core API, recorrendo à linguem de programação C++, para a DQC que permita a comunicação entre os serviços *Cloud* e os dispositivos, conferindo a estes um conjunto de operações e uma ilimitada lista de serviços.

Com esta API qualquer integrador, desenvolvedor ou cliente fica com a possibilidade de comunicar e de executar um conjunto de operações e serviços sob os seus dispositivos *IoT* inteligentes. Uma das outras grandes vantagens desta API é a possibilidade de o cliente puder produzir a sua própria solução de acordo com as suas necessidades e requisitos, mas integrando a API, uma vez que esta é genérica, permitindo a sua integração em vários tipos de ecossistemas.

A comunicação com esta API é feita por pedidos REST, sob o protocolo WEB, onde o formato de mensagens usado durante a comunicação é o JSON. Deste modo, foi também necessário desenvolver uma *RESTful* API que desse suporte ao *website* desenvolvido - Quantum Portal, recorrendo à linguagem de programação em Java, permitindo a comunicação deste com a core API presente na DQC, tendo assim acesso a alguns dos serviços e funcionalidades fornecidos por esta.

O protocolo de comunicação usado foi o Advanced Message Queuing Protocol (AMQP), sendo o RabbitMQ o *message broker* utilizado para dar suporte ao AMQP.

1.2 Entidade de acolhimento

Domatica Global Solutions, S.A. fundada no ano de 2002 é uma empresa tecnológica especializada em soluções de engenharia. Graças à sua equipa altamente qualificada e

sofisticada, o seu principal objetivo passa pelo desenvolvimento de uma robusta *framework* para soluções Machine-to-Machine(M2M) /IoT.

A Domatica Global Solutions é criadora de um *middleware* IoT, Domatica Quantum System (DQS). O DQS é uma Application Enablement Platform (AEP) que permite o rápido desenvolvimento para soluções M2M/IoT. Esta *framework* permite que qualquer utilizador, sem qualquer tipo de experiência e/ou conhecimento aprofundado, possa desenvolver de uma forma bastante simples, comunicações entre dispositivos e aplicações de *software*. Deste modo, a *framework* IoT desenvolvida pela Domatica é adequada para desenvolvedores de *software*, fabricantes de *hardware* e integradores de sistemas que procuram oferecer serviços, soluções e novos modelos de negócio em variadas áreas, tais como Agricultura, Automação de Edifícios, Gestão de Energia, *Smart Cities* ou Indústria 4.0. Esta tecnologia já se encontra em utilização em diversas entidades espalhadas um pouco por todo o mundo, como é o caso de Portugal, Espanha, Irlanda, Itália, Reino Unido, África do Sul, China, Brasil, México e Estados Unidos da Améria [4].

Neste momento a Domatica conta com uma diversificada rede de parceiros nacionais e internacionais, a nível de *software*, hardware e fornecedores de serviços, estando cada vez mais a aumentar a sua rede e o volume de negócios.

Mais recentemente a Domatica tornou-se parceira a nível de *hardware* de algumas empresas muito conceituadas a nível internacional, destacando-se a Cisco, Dell, Intel, Huawei ou Hewlett-Packard Company, vulgarmente conhecida como HP.

1.3 Motivação e objetivos do projeto

O mundo da *IoT* tem sofrido grandes avanços tecnológicos. Segundo um relatório de mobilidade levado a cabo pela Ericsson, entre 2015 e 2021 o número de dispositivos ligados à *IoT* irá crescer em cerca de 23%, prevendo-se que já em 2018 o número de dispositivos *IoT* será superior ao número de telemóveis, tendo em conta a categoria de dispositivos ligados a uma rede [5].

A *IoT* traz várias vantagens para o utilizador, tornando certos objetos do seu quotidiano mais inteligentes, facilitando-os assim o seu dia-a-dia. No entanto também no mundo da indústria a *IoT* apresenta um papel importante, tornando por exemplo mais eficiente certos processos em linhas de montagem, ou num controlo mais eficiente nos gastos elétricos de uma fábrica.

A Domatica Global Solutions sentiu a necessidade de reformular a Core API da DQC de modo a satisfazer a sua necessidade e dos seus clientes, tornando a sua tecnologia mais robusta e funcional permitindo que esta acompanhe o crescente avanço tecnológico nesta área.

Deste modo, o objetivo deste projeto passava pela criação de uma Core API para a Domatica Quantum Cloud (DQC). A DQC é um conector *IoT* que permite a comunicação entre os serviços presentes na *cloud* e os dispositivos de campo de forma integrada através de um abrangente ecossistema de produtos, serviços e protocolos, permitindo também a integração com outro tipo de produtos. Esta solução é adequada para integradores, provedores de soluções e desenvolvedores, permitindo comunicação em real time entre a *Cloud* e os dispositivos e ainda a virtualização de qualquer dispositivo, entre diversas tecnologias, conferindo lhes uma ilimitada lista de serviços [4].

Por outro lado, foi necessário desenvolver também uma *RESTful* API, que tinha como base servir o Quantum Portal. Assim o objetivo desta *RESTful* API era o de promover a comunicação com a Core API, uma vez que não existe comunicação direta com a Core API. A Core API responde por pedidos REST, dai a necessidade da criação de uma *RESTful* API que servisse como base de comunicação para o envio de mensagens e a respetivas resposta, tendo como base o JSON para a representação do formato dessas mesmas mensagens e respostas. Sendo a Core API, uma API genérica, é possível desenvolver diferentes *RESTful API's* que comuniquem com as diferentes funcionalidades fornecidas pela Core API.

A Core API disponibiliza algumas funcionalidades principais. Um dos objetivos é promover a comunicação com os dispositivos *IoT* inteligentes de modo a que fosse possível controla-los (por exemplo acender e/ou apagar uma luz), ter acesso aos dados por eles lidos (por exemplo um sensor de temperatura ou umidade), permitir a sua configuração e também ter acesso outro tipo de serviços por eles disponibilizados. Muitos dos recursos gerados estão a operar em *runtime* nos dispositivos e na *Cloud* ou guardados numa base de dados MySQL.

Todo o esquema arquitetural, tecnologias usadas e funcionalidades implementas e seu funcionamento, relativamente aos DQNs e à DQC, bem como todos os aspetos mais detalhados serão explicados mais aprofundadamente no Capítulo 4 – Arquitetura.

1.4 Estrutura do relatório

O presente relatório é constituído por vários capítulos, sendo os mesmos organizados com a seguinte estrutura:

- **Capítulo 1** – Introdução, onde é realizada uma breve abordagem ao tema da *Internet of Things* e onde é feito um enquadramento do projeto relativamente aos objetivos propostos; Entidade de acolhimento, onde é feita uma descrição da entidade de acolhimento; Motivação e Objetivos do Projeto, onde é descrito com mais pormenor qual a motivação e os objetivos propostos;
- **Capítulo 2** – Estado da Arte, onde se encontram descritas algumas aplicações e/ou tecnologias semelhantes às tratadas neste documento;
- **Capítulo 3**– Metodologia e Gestão de Projeto, onde são explicados os métodos de desenvolvimento e quais as ferramentas utilizadas durante a gestão do projeto;
- **Capítulo 4** - Arquitetura, onde é descrita a arquitetura usada, a sua estrutura bem como os protocolos de comunicação usados;
- **Capítulo 5** – Implementação, Testes e Segurança, onde são referidas e explicadas todas as funcionalidades implementadas, testes feitos à solução bem como ferramentas utilizadas em auxílio para a realização dos mesmos e alguns aspetos a nível de segurança que foram tidos em conta durante a implementação;
- **Capítulo 6** – Conclusão, onde é apresentada uma reflexão do trabalho realizado, quais os objetivos cumpridos, os conhecimentos adquiridos, dificuldades encontradas e também uma referência a possível trabalho futuro a desenvolver.

2. Estado da arte

Antes do desenvolvimento de qualquer solução, existem certos aspetos a ter em consideração. Um desses aspetos é a análise e pesquisa de tecnologias, serviços, aplicações ou projetos existentes na área para a qualquer se vai desenvolver o novo projeto. Com esta

pesquisa e análise é possível ter uma visão geral do que já existe implementado e em funcionamento, para identificar possíveis vantagens e desvantagens de cada uma dessas soluções, e assim servir de suporte ao desenvolvimento de uma nova solução.

No Capítulo 2.1 Web Services é feita uma revisão literária sobre Web Services, onde são apresentadas algumas das tecnologias usadas, bem como a comparação entre os tipos de Web Service mais usados.

No Capítulo 2.2 são apresentados algumas aplicações e serviços que usam na sua arquitetura algumas das tecnologias usadas neste projeto.

No Capítulo 2.3 é feita uma abordagem ao Advanced Message Queuing Protocol (AMQP), uma vez que foi o protocolo de comunicação usado pelo *broker* RabbiMQ, para as diversas trocas de mensagens.

2.1 Web Services

Segundo a W3C [6], um *Web Service* é sistema de *software* desenhado para suportar comunicações entre duas máquinas, promovendo a comunicação entre diferentes aplicações. Ao promover esta comunicação, é possível interconectar aplicações existentes em diferentes sistemas operativos e/o desenvolvidas com linguagens de programação distintas. Os Web Services foram projetados para promover essa comunicação sob a internet, usando para isso a *Uniform Resource Identifier* (URI). A URI é uma cadeira de caracteres para a representação de recursos, sejam eles abstratos ou físicos [7].

Atualmente os *Web Services* são um dos métodos mais usados para promover a comunicação entre sistemas, utilizando o XML, JSON, CSV, entre outros como formato de mensagens.

2.1.2 Representações Web Service

A arquitetura dos *Web Services* pode ser representada de diferentes maneiras, destacando-se o *Service-Oriented Architecture* (SOAP) e o *Representational State Transfer* (REST). De seguida serão abordadas algumas dessas diferentes maneiras de representação de um *Web Service*.

- ***Remote Procedure Call*** **(RPC)**, define-se como uma tecnologia de comunicação que permite a chamada de um procedimento que exista noutra máquina, usando para isso a comunicação via internet [8].
- ***Web Services Description Language*** **(WSDL)**, descreve um Web Service baseado num formato XML. O WSDL descreve os serviços, as mensagens, as operações, e os URL para a chamada dos serviços, independentemente dos protocolos de rede usados [9].
- ***Simple Object Access Protocol*** **(SOAP),** é um protocolo que especifica a invocação de métodos existentes em servidores, serviços, componentes e objetos. É também uma maneira de criar ambientes de computação complexa, amplamente distribuídos, utilizando a infraestrutura de internet existente. Com o SOAP as aplicações existentes em plataformas heterógenas, podem comunicar entre si [10]. Utiliza o XML como formato de mensagens.
- ***Service-Oriented Architecture*** **(SOA)** – Os *Web Services* conseguem ser implementados de acordo com os conceitos arquiteturais SOA. SOA define que as funcionalidades implementadas pelas aplicações, devem ser representadas na forma de serviços. Esta arquitetura define um modelo de interação em três principais partes. O provedor de serviços, que descreve o serviço e fornece implementação para o mesmo, o consumidor do serviço, que pode usar o URI como de descrição do serviço, e por último o *service broker*, que é responsável por promover e manter o serviço registado [11].
- ***Windows Communication Foundation*** **(WCF)** – é uma plataforma da Microsoft usada para promover a comunicação entre sistemas e para a construção de aplicações orientadas a serviço. É uma plataforma simples de usar, robusta e de fácil integração. Com o WCF é possível enviar dados, como mensagens assíncronas, de um ponto para outro. Um cliente WCF conectasse a um serviço WCF através de um ponto. Esse ponto é um URL que especifica onde esse mesmo ponto quer aceder, de modo a estabelecer a comunicação.
- ***Representational State Transfer*** **(REST)** – O termo REST foi definido por Roy Fielding na sua dissertação [12]. REST é um estilo arquitetónico para a construção de sistemas distribuídos. Usa um conjunto de restrições para estabelecer uma relação cliente/servidor e está usualmente relacionado ao protocolo HTTP. Existem alguns princípios associados ao REST. Um deles é

a utilização de recursos para a estruturação dos URIs. Um recurso é usualmente algo que pode ficar guardado num computador. A comunicação em REST é feita através da transferência de representações desses mesmos recursos. Outro dos princípios é o *stateless*. Isto significa que os estados da sessão resultantes da comunicação entre cliente e servidor, são guardados no cliente e não no servidor. Deste modo durante a comunicação é sempre enviada toda a informação necessária. Isto traduz-se em algumas vantagens. Por um lado, o processamento no servidor é mais baixo, uma vez que durante o pedido efetuado é enviada toda a informação necessária. A capacidade de recuperação de falhas é melhorada, não existindo também a necessidade de manter o estado dos pedidos, permitindo que o servidor possa limpar os recursos de forma mais rápida [12]. Usualmente as tradicionais aplicações web permitem apenas a comunicação usando operações HTTP GET e POST. Com a introdução dos conceitos REST é possível usar o protocolo HTTP operações CRUD (*Create, Read, Update, Delete*). Estas operações permitem a utilização de outros métodos, além do GET (receber um recurso) e POST (criar um recurso), como é o caso do PUT (atualizar um recurso) e do DELTE (eliminar um recurso).

2.2 Aplicações e Serviços

Neste capítulo serão descritas algumas aplicações e serviços que de alguma forma se assemelham à área de desenvolvimento deste projeto.

2.2.1 Azure IoT Hub

O Azure IoT Hub é um serviço proprietário da Microsoft Azure. Este serviço permite comunicações bidirecionais e seguras entre vários dispositivos IoT, uma solução *back-end*, onde existem também comunicações entre os dispositivos e a *cloud* e vice-versa, o controlo de acessos através de chaves de segurança e gestão e monitorização da conectividade do dispositivo e dos seus eventos. Na Figura 1 - Arquitetura Azure IoT Hub é possível observar a arquitetura deste serviço [13].

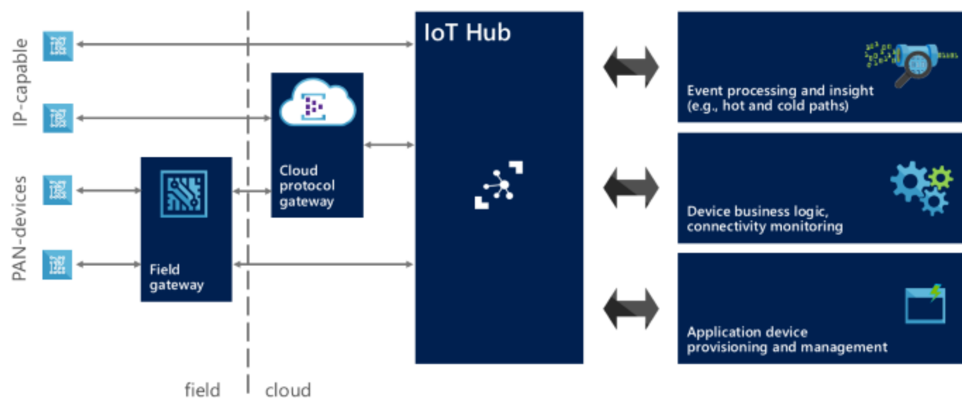

Figura 1 - Arquitetura Azure IoT Hub

O IoT hub tenda em conta um conjunto especificações relativamente aos dispositivos IoT. Os mesmos tratam-se frequentemente de sistemas integrados sem qualquer tipo de controlo humano, podendo os mesmos estarem em localizações remotas, onde não é possível a existência de acesso físico. Estes dispositivos podem ser acedidos e controlados através de uma solução *back-end,* no entanto podem apresentar uma conectividade à rede lenta e dispendiosa. Para as comunicações bidirecionais, este serviço suporta o protocolo HTTP e o AMQP. Deste modo qualquer dispositivo que suporte IP (*Internet Protocol*) pode comunicar com o Hub para o envio e receção de informações. No entanto para alguns dispositivos, os protocolos HTTP e AMQP não são suficientes. Nestes casos, o Azure dispõe de uma *framework* – Azure IoT *protocol* -, para a adaptação de protocolos, de modo a que os dispositivos possam comunicar com o Hub, bem como um vasto conjunto de SDKs que têm suporte para várias plataformas e linguagens de programação. [14].

2.2.2. RoboMQ

O RoboMQ Figura 2 - RoboMQ é uma plataforma de *middleware* para a IoT disponível para a *cloud*. O seu principal objetivo é de conectar dispositivos, sistemas e *cloud* para o desenvolvimento de aplicações inteligentes e sensíveis ao contexto. É uma

plataforma que funciona de modo a que qualquer dispositivo possa comunicar com qualquer aplicação.

As componentes do *core* desta plataforma foram projetadas para uma completada gestão de componentes, monitorização alertas. Fornece ainda um *hub* integrado para filas de mensagens. Destacam-se ainda outras características desta plataforma. Entre elas, permite uma entrega de mensagens confiável, ou seja, possibilita que as mensagens sejam persistentes e duradoras; tem suporte para vários protocolos de comunicação, entre eles, o MQTT, AMQP, STOMP, HTTP e REST; suporte para várias linguagens de programação, garantindo também uma conexão segura através de certificados SSL. A sua interface gráfica permite a administração de todas as definições desta plataforma. Possibilita a gestão de comunicações, produtores e consumidores das listas de esperas, bem como adicionar e gerir o acesso de diversos utilizadores. A sua *dashboard* permite visualizar em tempo real, informação estatística, número de conexões, mensagens em lista de espera e entregues. O seu SDK permite o desenvolvimento para diversas linguagens de programação [15].

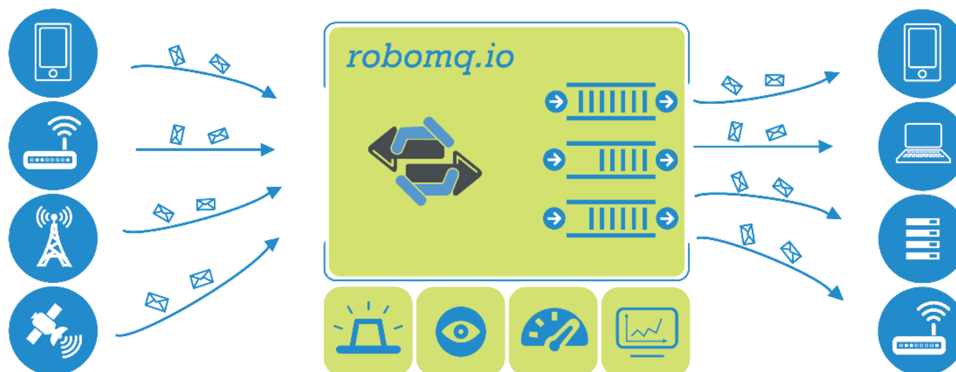

Figura 2 - RoboMQ

2.2.3 DeviceHive

DeviceHive é uma *framework* que fornece uma base sólida de desenvolvimento para criar ou desenvolver soluções IoT e *machine-to-machine* (M2M), superando assim a grandes distância e complexidade entre o desenvolvimento integrado, plataformas em *cloud*, grandes quantidades de dados e aplicações clientes [16].

Esta *framework* possibilita a integração com outras *clouds* ou plataformas, onde é permito a inclusão da Amazon Alexa. A Amazon Alexa, é uma aplicação móvel para iOS E Android, onde desempenha um papel de assistente virtual inteligente, tendo sido desenvolvida pela Amazon. Este assistente virtual possui a capacidade de integrar voz, tocar músicas, configurar alarmes, bem como adquirir informações de tráfego, temperatura, entre outros [17]. Permite também controlar dispositivos IoT. Por outro lado, com o DeviceHive podemos incluir um painel de visualização totalmente personalizável, onde facilmente é possível construir dados analíticos, incluir várias fontes de dados diferentes para processamento em tempo real, entre eles o Apache Spark, Cassandra e ElasticSearch [16].

Em termos de conexão é possível comunicar com qualquer dispositivo, através de uma REST API - Figura 3 - REST API (DeviceHive), WebSockets, ou pelo protocolo MQTT. Tem suporte para várias linguagens de programação (Phyton ou Java por exemplo) e para plataformas como iOS ou Android. É também possível conectar-se por WI-FI de baixa qualidade, como é o por exemplo o módulo ESP8266 [16].

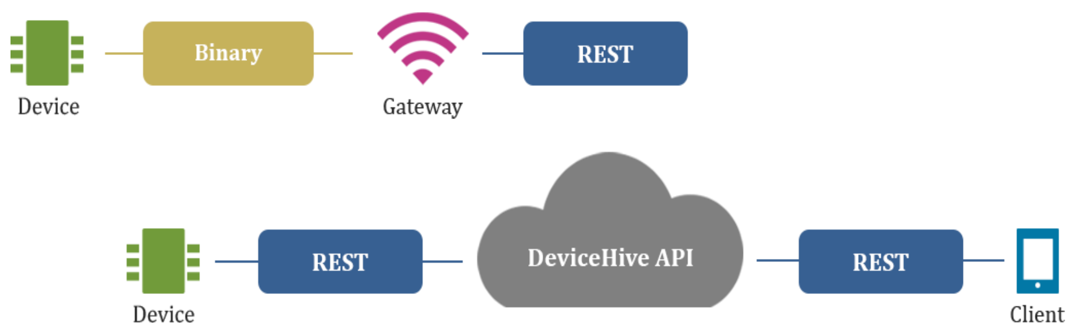

Figura 3 - REST API (DeviceHive)

2.2.4 macchina.io

A macchina.io é conjunto de ferramentas *software* para o desenvolvimento rápido de aplicações para dispositivos IoT. Estes dipositivos funcionam em sistemas baseados em Linux (Raspberry pi, Beaglebone, MangOH). Oferece também um ambiente para desenvolvimento em C++, Javascript, fornecendo também blocos de código para

desenvolvimentos mais rápidos e fáceis. Estas aplicações desenvolvidas têm a possibilidade de comunicar com diversos sensores e serviços em *cloud*, onde existe processamento, análise e filtragem dos dados proveniente desses sensores e/ou serviços. As suas principais vantagens são a rápida criação de aplicações para dispositivos inteligentes, permitindo o desenvolvimento em diferentes linguais de programação, é bastante otimizada, funcionando por exemplo em dispositivos Linux com 32 MB de memória Ram [18].

A macchine.io é essencialmente para fabricantes de dispositivos e integradores de sistemas, uma vez que fornece uma estrutura de *software* escalável e também com atualizações regulares [18].

Permite comunicação com serviços cloud através de HTTPS ou REST API e com os diversos dispositivos utiliza o CoAP, Bluetooh, USB, Modbus, entre outros -, Figura 4 - Comunicação.

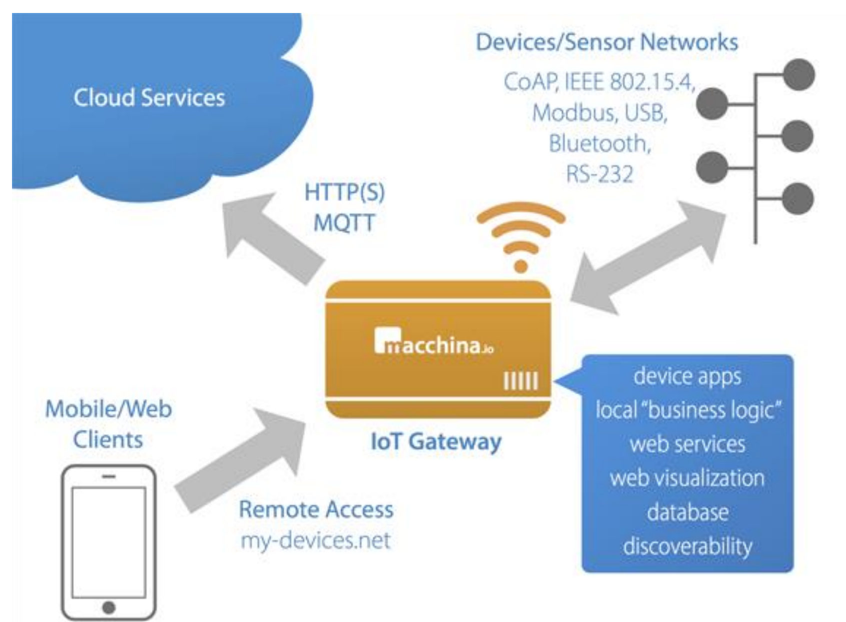

Figura 4 - Comunicação

2.2.5 Mainspring

A Mainspring é uma *framework* que permite a construção de soluções M2M, tais como o *tracking* de veículos, ou monotorização remota. Neste tipo de aplicações,

geralmente os dispositivos tem um conjunto de sensores a eles ligados, como por exemplo gps, temperatura e outros. Esta plataforma toma conta de toda a comunicação com os *devices*, a sua configuração, bem como o armazenamento de diferentes tipos de dados, permitindo assim que os desenvolvedores tenham apenas a preocupação na porta lógica do negócio. Esta plataforma pode estar conectada a diversos dispositivos conhecidos, tais como o Raspberry Pi, Arduino, smartphones com sistema operativo Android ou ao Ubuntu Cor. A comunicação com os dispositivos inteligentes pode ser feita através de uma REST API, ou através de mensagens HTTP GET ou POST, abrangendo assim as principais funcionalidades comuns e exigidas numa solução M2M [19].

2.2.6 Node-RED

O Node-RED é uma plataforma de programação *online* desenvolvida em 2013 e que é compatível com vasto conjunto de dispositivos, entre eles o Arduino, Raspberry Pi, Beaglone e Intel Edison. Esta plataforma tem como base o Node.js. O Node.js é um interpretador de código JavaScript, permitindo a criação de soluções de alta escalabilidade. Nesta plataforma, a sua configuração é maioritariamente visual, facilitando assim a integração dos vários dispositivos. Para a comunicação com os diversos dispositivos, o Node-RED utiliza vários protocolos conhecidos, como por exemplo TCP, UDP e MQTT. Outras das vantagens é a possibilidade de integrar redes sociais e o mail e assim com recursos a ambas controlar por exemplo uma lâmpada. Esta plataforma oferece um editor no *browser*, para a construção de fluxos de código, podendo ser usados para o desenvolvimento de funções em JavaScript. Estes fluxos de código, alguns deles assinalados a vermelho na Figura 5 - Fluxos de código (Node-RED) -, podem incluir funções JavaScript com diferentes comportamentos. Cada fluxo de código pode ser interligado a outro e assim sucessivamente, de modo a construir uma sequência lógica, em que certos fluxos de código só são executados, após a execução de outros fluxos de código. Estes fluxos trazem algumas vantagens no desenvolvimento de soluções, uma vez que de forma simples e rápida é possível desenvolver várias funções diferentes e fazer reutilização dessas mesmas funções, em que um só fluxo pode estar interligado a mais do que um outro fluxo de código, permitindo também o desenvolvimento social. Isto significa que é

possível importar e exportar diferentes tipos de fluxos, estando os mesmos guardados em formato JSON [20].

Figura 5 - Fluxos de código (Node-RED)

As soluções desenvolvidas nesta plataforma podem ser executadas localmente, no próprio dispositivo ou na *cloud*, uma vez que têm suporte para vários serviços *cloud* tais como a Amazon Web Service e a Microsoft Azure.

Atualmente é utilizado por diversos tipos de *software* IoT, onde recentemente a Domatica Global Solutions foi uma das empresas a integrar esta plataforma no seu ecossistema.

2.2.7 Open Connectivity Foundation

A Open Connectivity Foundation (OCF) é um grupo industrial onde o seu principal objetivo passa por desenvolver padrões de conectividade para dispositivos IoT. Este grupo nasceu da necessidade de não haver nenhum idioma universal para o mundo IoT. Os fabricantes dos dispositivos viram-se obrigados a escolher *frameworks* já existentes no mercado, como é o caso da Amazon ou Google, limitando assim o desenvolvimento em diferentes tipos de ecossistemas. Este facto dificultava a utilização por parte dos utilizadores finais, que se viam obrigados a determinar se os produtos que utilizavam eram

compatíveis com os ecossistemas que utilizavam ou encontrar maneiras de integrar os seus dispositivos [21].

Deste modo, a OCF fornece especificações, código e um programa de certificações para que os fabricantes possam desenvolver produtos certificados e compatíveis com esta solução, de modo que possam operar com dispositivos IoT. Pretende também melhorar a experiência dos utilizadores, uma vez que facilita a passagem dos seus dispositivos para diferentes tipos de ecossistemas. As certificações da OCF utilizam padrões e tecnologias já existentes no mercado, fornecendo mecanismo de conexão entre dispositivos e também entre dispositivos e a *cloud*, gerindo esse fluxo de informação independentemente dos sistemas ou prestadores de serviços [21].

A IoTivity é uma *framework open source*, que permite uma conectividade com as necessidades emergentes do mundo da IoT, utilizando para isso os vários recursos disponibilizados pela OCF. Esta *framework* disponibiliza um conjunto de APIs e ferramentas para diversas linguagens de programação, entre elas o Java, C e C++.

2.2.8 Zetta

A Zetta é uma plataforma *open source* para a criação de servidores IoT, desenvolvida com base em Node.js, REST, WebSockets e programação reativa. A Programação reativa baseia-se em fluxos de dados assíncronos. Quer isto dizer, que ao contrário da forma tradicional de programar e desenvolver, com o uso de programação reativa, não existe uma ordem cronológica e linear para o desencadeamento das diversas ações, podendo algumas delas serem desencadeadas independentemente se encontra falhas ou não durante o seu percurso. Isto torna a solução mais inteligente, permitindo que as mesmas possam gerir as suas falhas, construir rotinas diferente e estar sempre online, traduzindo-se numa mais-valia para os dispositivos inteligentes existentes no mundo IoT. Os servidores Zetta podem ser executados tanto na *cloud*, em computadores ou noutros dispositivos tais como o Raspberry Pi. Por outro lado, a Zetta transforma qualquer dispositivo numa API – Figura 6 - Topologia Zetta, fornecendo a estes uma API REST que pode funcionar localmente ou na *cloud*. Atualmente a construção de soluções IoT ainda é um processo algo complexo. No entanto com esta plataforma essa construção é bastante simplificada, devido às suas abstrações e protocolos fornecidos [22].

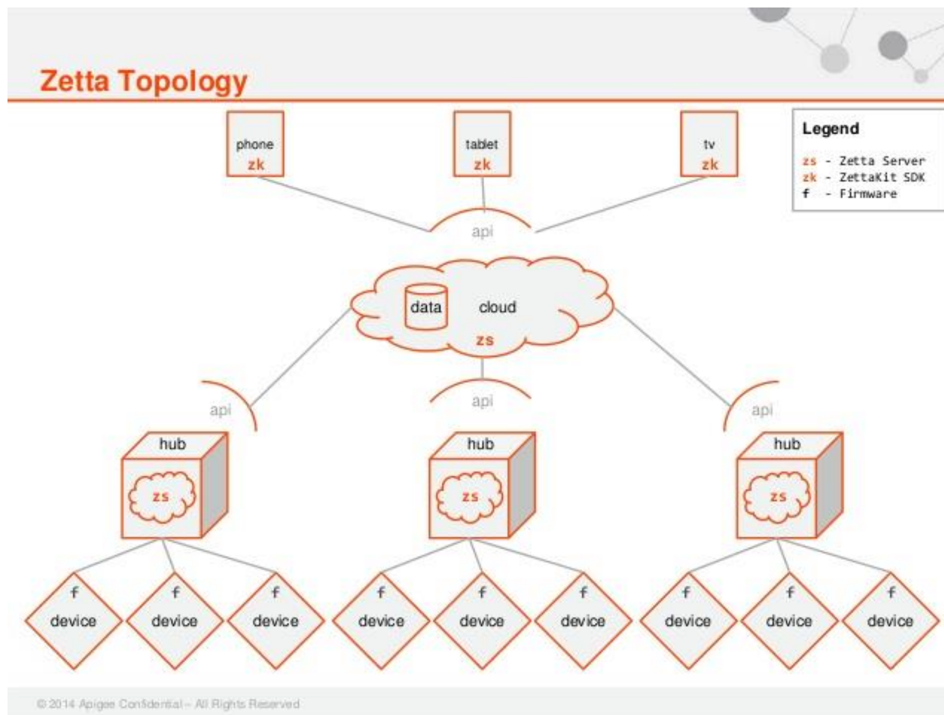

Figura 6 - Topologia Zetta

2.2.9 ThingWorx

A Thingworx, pertencente à empresa PTC, é uma plataforma desenhada especificamente para a IoT. Os seus utilizadores têm a possibilidade de conectar, criar e desenhar soluções IoT inovadores. O seu principal objetivo é facilitar o desenvolvimento de soluções de forma a que os dispositivos de diferentes fabricantes possam comunicar entre si e partilhar dados.

Esta plataforma disponibiliza um conjunto de ferramentas tornando-se atualmente uma das maiores plataformas de desenvolvimento para soluções IoT. A ThingWorx opera com base em alguns conceitos, como é possível observar na Figura 7 - Conceitos Thingworx.

Figura 7 - Conceitos Thingworx

Em termos de segurança permite a integração de utilizadores e grupos, bem como a comunicação entre sistemas, pessoas e persistência de dados. Oferece também em termos de experiência uma ferramenta de *design* de modo a que seja possível criar toda a solução em termos de *fron-end*, seja ela em ambiente mobile ou web. Para o desenvolvimento, utiliza ferramentas de *drag-and-drop* de modo a facilitar o desenho do modelo de dados, definir a arquitetura por de trás da solução, permitindo a integração com diferentes plug-ins. Disponibiliza também ferramentas de análise ao processar todos os dados provenientes dos diferentes dispositivos. Destas ferramentas, destacam-se diferentes tipos de visualização como por exemplos gráficos. Em termos de comunicação, fornece um editor de código *online* em Javascript, uma REST API, bem como o seu SDK proprietário, tornando assim esta plataforma mais versátil e compatível com diferentes ecossistemas e linguagens de programação, facilitando assim a comunicação. Por fim, permite também a gestão de dispositivos, acesso remoto, gestão de *software* bem como *troubleshooting* e *debugging* [23]. Na Figura 8 - Conectores ThingWorx é possível constatar o número de serviços e protocolos diferentes possíveis de conectar a esta plataforma.

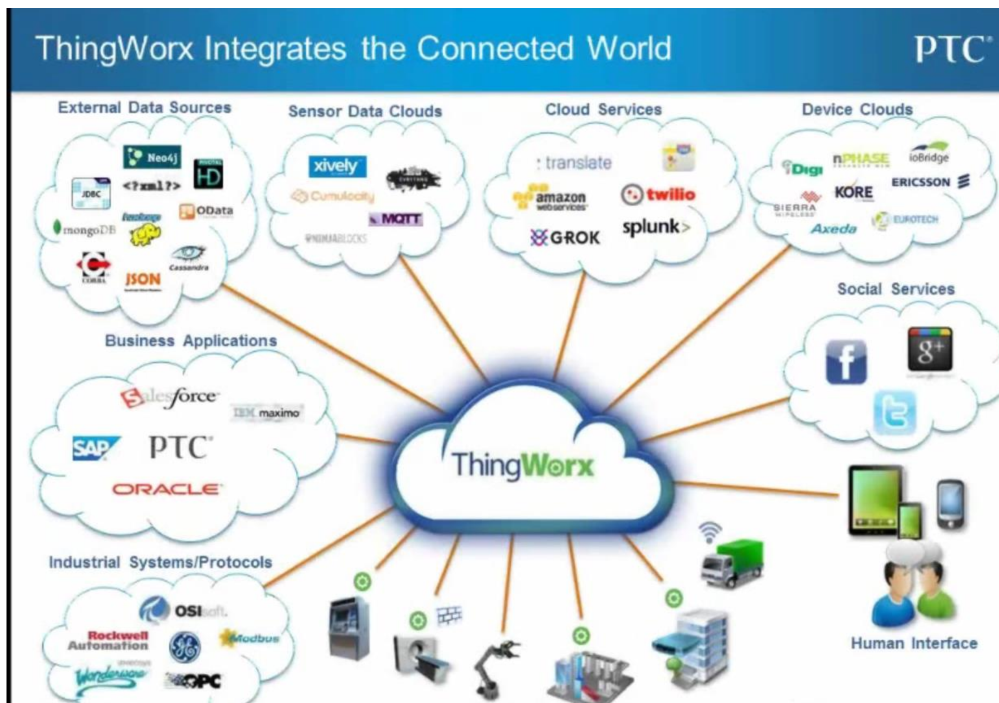

Figura 8 - Conectores ThingWorx

Muito recentemente, a entidade de acolhimento – Domatica Global Solutions -, participou no IoT Challenge (2ª edição) lançado pela Portugal Telecom, na qual fiz parte da equipa de desenvolvimento responsável por participar e desenvolver uma solução. O objetivo passou por integrar os serviços já existentes na Domatica e integra-los com esta plataforma. Este projeto simulava uma linha de produção industrial em tempo real, onde era possível fazer toda a monitorização (leitura de temperaturas, número de peças fabricadas, tempo de produção, etc), também o controlo (desligar ou ligar a máquina, etc), através da ThingWorx, bem como a implementação de alarmes e alertas, quando por exemplo uma peça ficava presa na linha de produção ou quanto a temperatura subia acima de determinado valor (por exemplo 30º celsius). Deste evento, fomos um dos vencedores eleitos por um conjunto de júris, desde responsáveis da PT como representantes da ThingWorx. A Figura 9 - IoT Challenge, retirada de um artigo do jornal dinheirovivo (https://www.dinheirovivo.pt/economia/machine-to-machine-a-tecnologia-que-da-vida-a-coisas/), mostra o momento da entrega do prémio, onde à esquerda se encontra o Doutor Abel Aguiar, diretor de produto e pré-venda da Portugal Telecom, acompanhado por uma parte da equipa de desenvolvimento da Domatica.

Figura 9 - IoT Challenge

2.2.10 IoT real time data acquisition using MQTT protocol

Este projeto tinha como objetivo a recolha de dados em real time, temperatura e humidade, usando o MQTT como protocolo de comunicação. Estes foram posteriormente guardados numa base de dados MySQL. O MQTT é um protocolo desenhado especificamente para aplicações *machine-to-machine* que opera sob o TCP/IP, utilizando uma largura de banda bastante baixa, o que torna este protoloco ideal para soluções IoT. Este protocolo permite ainda a transmissão de dados em diversos formatos, como por exemplo, em formato binário, texto, XML ou JSON, usando um modelo de *publish/subscribe* [24].

Um sistema que use o protocolo MQTT necessita essencialmente de duas componentes. Um cliente MQTT, que se encontra instalado no dispositivo. Este cliente pode por exemplo, ser uma plataforma web (que use Javascript), podendo para isso usar a biblioteca *Client* PAHO do Eclipse. Um *broker* MQTT, responsável por publicar e subscrever os dados. Neste caso pode ser usado uma plataforma Linux, utilizando um *broker* free, como é o caso do mosquitto, HiveMQ, entre outros [24].

Para a implementação deste projeto, foram usados alguns componentes. Foi usado um módulo Wi-fi (Wemos D1 Mini & ESP 8266 Wi-fi Module), um sensor de temperatura

& humidade (DHT11 Temperature & Humidity Sensor), um computador que desempenhava o papel de *broker*, e um smartphone Android para a leitura dos dados recolhidos. Os dados recolhidos pelo sensor DHT11 eram enviados (*publish*) para o *broker* MQTT, sendo posteriormente subscrita (*subscribe*) pela aplicação cliente. Os dados após serem recolhidos eram guardados na base de dados MySQL, e apresentados na interface gráfica da aplicação cliente [24].

2.2.11 Sense Tecnic Systems

A Sense Tecnic Systems é uma companhia de *software* IoT que ajuda os seus clientes a desenvolver e integrar avançadas soluções IoT. Tem elevados conhecimentos técnicos numa variedade de tecnologias IoT, através do uso de plataformas industriais de gestão e ferramentas de gestão avançadas. Para o seu *core* utilizam o Node-RED para o desenvolvimento e integração de aplicações IoT. Utiliza também o FRED. O FRED é uma plataforma para o desenvolvimento de soluções. Assim todo a programação visual é efetuada no Node-RED, onde o FRED tem o papel de promover um ambiente de hospedagem confiável, fácil acesso tornando assim o desenvolvimento de soluções bastante mais simples e rápidas. As principais características do FRED é a rapidez, a simplicidade de construção, flexibilidade, confiável uma vez que uso a Amazon AWS Cloud. Recentemente e de forma a satisfazer as necessidades dos seus clientes, foi desenvolvida um painel com sistemas de monitorização onde é possível conectar o FRED com o protocolo de comunicação MQTT. Assim cada cliente posse visualizar os dados no seu painel, permitindo o acesso a dados em tempo real e a histórico de dados. Permite ainda a configuração de alertas e diferentes tipos de visualização, onde os fluxos desenvolvidos no Node-RED, adaptam-se a todos esses tipos de visualização [25].

2.3 Advanced Message Queuing Protocol (AMQP)

O AMQP é um protocolo *middleware* para o padrão de mensagens. Qualquer produto *middleware* é desenvolvido para diferentes plataformas e linguagens de programação, permitindo assim a troca de mensagens. Atualmente o AMQP é já usado por várias empresas e sistemas, onde se destaca a Cisco System, Credit Suisse, Red Hat, entre outras [26].

O AMQP permite que as aplicações possam enviar e receber mensagens, funcionando por exemplo como serviço de *email*. O AMQP diferencia-se de outras soluções já existentes, uma vez que permite a especificação das mensagens a receber e a enviar e como essa troca de mensagens pode ser efetuada em termos de segurança, desempenho e confiabilidade. O AMQP tem certos aspetos que funcionam muito melhor do que outros protocolos semelhantes. É seguro, possibilita a conexão de aplicações existentes em diferentes plataformas e a conexão de parceiros de negócios, usando para isso o padrão e a inovação existentes nas bases deste protocolo. As especificações deste protocolo ainda não se encontram totalmente finalizadas, no entanto já existem vários projetos que usam o AMQP, como por exemplo o Red Hat, VMware [26].

3. Metodologia e Gestão de projeto

Neste capítulo será explicada a metodologia de desenvolvimento utilizada durante o desenvolvimento do projeto, bem como a ferramenta de gestão de projeto utilizada para a monitorização das tarefas da equipa de desenvolvimento.

3.1 Metodologia

O sucesso de um projeto depende de variados fatores, sendo a metodologia de desenvolvimento um desses fatores. Para o desenvolvimento deste projeto a metodologia escolhida foi a metodologia ágil SCRUM. As metodologias ágeis têm como objetivo acelerar o desenvolvimento de *software* ao mesmo tempo que trazem benefícios ao nível da comunicação e interação da equipa de desenvolvimento.

A metodologia ágil SCRUM apresenta algumas características principais, entre elas o desenvolvimento iterativo e incremental, flexibilidade na definição de requisitos e comunicação/colaboração entre o cliente e equipa de desenvolvimento. Nesta metodologia, os projetos são divididos em ciclos a que se dá o nome de sprints. Um sprint representa um conjunto de tarefas, dentro de um determinado tempo, para serem executadas. Esta metodologia apresenta 3 principais papéis.

O primeiro é o Product Owner que é responsável pela definição de requisitos, datas e conteúdo para cada *release*. Pode também mudar os requisitos de cada sprint ficando também responsável por aceitar ou rejeitar o resultado da mesma.

Por sua vez o Scrum Master, é responsável por garantir o correto funcionamento da equipa de desenvolvimento, participando também nas reuniões diárias de forma acompanhar o planeamento e assim garantir que todo o processo está a ser encaminhado.

Já a equipa de desenvolvimento é responsável pelo desenvolvimento do projeto tendo em conta os requisitos e tarefas definidas.

Figura 10 - Metodologia Scrum

Figura 10, estão representadas as principais fases da metodologia Scrum. Na primeira fase, Produt Backlog, corresponde à lista de funcionalidades desejadas a serem implementadas pela equipa de desenvolvimento. Ao se iniciar cada *sprint*, é realizada uma reunião de planeamento, também conhecida por Sprint Planning Meeting, onde o Product Owner prioriza as tarefas e onde se define quais serão executadas durante esse sprint, o que corresponde à fase – Sprint Backlog. Usualmente durante o desenvolvimento de uma *sprint*, são realizadas curtas reuniões diárias entre a equipa de desenvolvimento de modo a que cada elemento dê conta do estado das suas tarefas e assim possibilitar a toda a equipa de desenvolvimento a ter um conhecimento do estado do projeto.

Para a realização desde projeto foram realizadas todas as segundas-feiras de cada semana reuniões com o responsável pelo projeto. Estas reuniões eram presenciais no escritório da empresa em Lisboa. É importante referir que este responsável pelo projeto, o CEO Domatica, fazia um pouco o papel de Produtct Owner como também de Scrum Master, uma vez que todas as definições de requisitos bem como o acompanhamento da

fase de desenvolvimento eram feitas pelo mesmo. O principal objetivo destas reuniões semanais era o de aferir o trabalho realizado na semana anterior, possíveis alterações de requisitos e/ou funcionalidades bem como definir novas tarefas a realizar posteriormente. Por outro lado, durante a semana eram também realizadas algumas reuniões, sempre que fosse necessário, para o esclarecimento de algumas dúvidas. Estas reuniões não eram presenciais, mas sim com recursos a algumas ferramentas de videoconferência, como é o caso do Skype ou do Cisco Spark.

3.2 Gestão de Projeto

Para a realização de projetos, principalmente se estes obrigarem a um elevado conjunto de tarefas, é conveniente recorrer a algum tipo de *software* ou ferramenta que auxilie a equipa de desenvolvimento na gestão destas mesmas tarefas. Neste projeto utilizamos o Asana, sendo esta uma plataforma com versão web e com aplicação móvel para Android e iOS. Esta plataforma possui uma coletânea de recursos, como por exemplo, lista de tarefas, criação de equipas, *chat* entre outros.

No Asana é possível criar várias equipas de desenvolvimento dentro da mesma organização, e deste modo atribuir projetos e responsáveis para tarefas específicas. Uma equipa de desenvolvimento tem assim acesso à lista de todas as tarefas, podendo criar novas secções e dentro de cada secção criar tarefas. Ao visionarmos a lista de tarefas podemos filtra-las pelas seguintes principais opções: Tarefas incompletas, tarefas completas, todas as tarefas, data conclusão e mostrar as tarefas atribuídas a cada membro da equipa de desenvolvimento. Para cada tarefa é possível realizar um conjunto de operações. É possível adicionar uma descrição, comentários, consultar quem criou a tarefa, qual o membro(s) a que essa tarefa esta designada e qual a data de conclusão. Temos também a possibilidade de adicionar *tags*, criar sub tarefas e adicionar ficheiros. Após a conclusão da tarefa, esta pode já pode ser marcada como realizada. Além da lista de tarefas, existe um *chat* onde a equipa de desenvolvimento pode conversar entre si e assim discutir tarefas ou criar novas ideias. O Asana oferece ainda um calendário onde é possível visionar a data de entrega das tarefas, cria gráficos onde é possível observar a evolução do trabalho da equipa de desenvolvimento, sendo também possível o agendamento de reuniões.

Neste projeto o Asana teve um papel principalmente ao nível da organização das tarefas. Foi criado nesta plataforma um projeto onde estava também incluída a equipa de

desenvolvimento responsável pelo o desenvolvimento do *web site* - Quantum Portal. Uma vez que o projeto desenvolvido por mim iria dar suporte a este Portal, era importante que tanto eu como a outra equipa de desenvolvimento tivéssemos acesso a ambas as tarefas. Isto permitia acompanhar o estado de desenvolvimentos de ambas as equipas.

Assim foram adicionadas no Asana um conjunto de várias tarefas, sendo a cada tarefa atribuído os responsáveis para a sua realização. Quando uma tarefa fosse executada e validada a sua execução pelo responsável superior, a mesma era marcada por realizada e assim sucessivamente.

3.3 Controlo de versões

Este projeto foi desenvolvido apenas por uma pessoa, no entanto era imprescindível o uso de um *software* que permitisse o controlo de versões bem como a realizações de backups dessas mesmas versões. Em primeiro lugar o controlo de versões tem como principal vantagem guardar o registo de todas as novas tarefas implementadas e adicionadas a cada *update* do repositório escolhido para guardar as diferentes versões. Deste modo era possível ter acesso a um histórico onde era permitido observar as tarefas que foram desenvolvidos. Por outro lado, com o uso de um *software* deste tipo era possível ter várias cópias diferentes do projeto nos seus diversos estágios de desenvolvimento. Isto permitia por um lado salvaguardar a existência de qualquer problema, havendo sempre acesso a uma cópia de segurança, bem como a possibilidade de voltar atrás num ponto de desenvolvimento, onde eram descartadas todas as alterações efetuadas posteriormente a esse ponto. Isto é uma mais-valia sempre que uma nova versão do projeto, por algum motivo em específico, se apresenta com problemas ou conflitos.

Para este propósito foi criado um repositório Git, hospedado nos servidores da empresa, onde eram efetuados *updates* a esse mesmo repositório sempre que necessário, salvaguardando assim todo o trabalho desenvolvido até aquele dado momento.

4. Arquitetura

Neste capítulo será explicado detalhadamente o que são os DQN, o seu funcionamento e tecnologias neles envolvidos. Será também explicado o funcionamento da DQC bem como toda a sua arquitetura e o modo como esta estabelece a conexão com os dispositivos e serviços.

A conceção da arquitetura antes da implementação é um fator crucial para o sucesso da implementação. Essa conceção inicial permite que durante a fase de implementação a mesma obedecesse a todos os requisitos arquiteturais definidos anteriormente, permitindo também que durante esta fase fosse já expectável ter conhecimento de estruturas, tecnologias, linguagens de programação, entre outros aspetos a serem usados, acelerando assim o projeto de desenvolvimento e de implementação.

Outro aspeto a ter em conta durante a conceção da arquitetura são os diferentes tipos de utilizadores finais que a API iria ter. É expectável que a maioria dos utilizadores desta API sejam equipas de desenvolvedores que necessitam de implementar esta mesma API nos seus softwares. Por outro lado, existem outro tipo de utilizadores, como por exemplo clientes, que poderão usar esta mesma API, no entanto não apresentam o mesmo conhecimento técnico que uma equipa de desenvolvedores. Deste modo, a criação de uma API em que a sua utilização fosse a mais simplificada possível era imprescindível. Outro facto, também ele imprescindível e de primordial importância, era a velocidade de processamento dos dados. Não era expectável por exemplo que um dado *request* a um sensor de temperatura demorasse por exemplo 1 ou 2 segundos, uma vez que a temperatura é um dado sensível que está constantemente em atualização e mudança de valores cada x milissegundos.

4.1 Domatica Quantum Nodes (DQN)

O DQN - Figura 11 - Domatica Quantum Node, é atualmente um Gateway IoT industrial desenvolvido pela Domatica Global Solutions [27].

Figura 11 - Domatica Quantum Node

O DQN funciona como um dispositivo *cloud*, instalado em campo, onde os vários serviços disponíveis na *cloud*, neste caso a DQC, são executados localmente. Esta interconexão entre DQN e DQC permite que o que acontece localmente, como por exemplo a leitura através de sensores de temperatura, CO_2 ou humidade, sejam apresentados na interface do utilizador em real time. Esta comunicação traz inúmeras vantagens, reduzindo a distância e complexidade entre os dispositivos e aplicações *software*, permitindo também o processamento de dados localmente, graças ao novo conceito de arquitetura IoT – *Fog Computing*[1].

O DQN é assim responsável pela comunicação entre vários protocolos de periféricos fornecendo dados precisos, confiáveis e significativos quer sejam em *cloud* ou localmente. Estes protocolos de periféricos são por exemplo KNX, Modbus ou Profinet [27].

O KNX é um padrão de protocolos de comunicação para controlo de residências e edifícios. Dado o exemplo de um edifício, existe cada vez mais a procura do aumento do conforto e versatilidade no controlo de sistema de ar condicionado ou de iluminação. Para isso é necessário que todos esses dados de controlo sejam transferidos para todos os componentes instalados responsáveis pela gestão do edifício. Deste modo o KNX tem como objetivo garantir que todos esses dispositivos comuniquem entre si através da mesma linguagem, o que reduz a complexidade de comunicação entre dispositivos de diferentes fabricantes, permitindo uma maior eficiência e uma redução de custos [28]. Por sua vez, o Modbus, criado pela Modicon, é um protocolo de comunicação de dados bastante utilizado em sistemas de automação industrial. É atualmente dos protocolos mais antigos e usados,

[1] Descentralização da computação para dispositivos mais próximos do utilizador

livre de licenciamento e fácil de instalar em diversos meios físicos. O Modbus apresenta alguns padrões físicos, tais como o RS-485 e Ethernet TCP/IP – Modbus TCP -, havendo variação de velocidade na comunicação padrão usado e também da extensão da rede e do número de dispositivos a ela ligados [29]. A Profinet é um padrão de comunicação Ethernet industrial sendo maioritariamente utilizado em automação industrial. Utilizando as vantagens do patrão Ethernet, a Profinet disponibiliza comunicação em tempo real, supervisão e monitorização.

Através do Quantum Node Extender e graças às interfaces de I/O do DQN, é possível interligar a este um vasto número de módulos externos, como é possível observar na Figura 12 - Quantum Node Extender.

A Domatica desenvolveu também um Software Development Kit

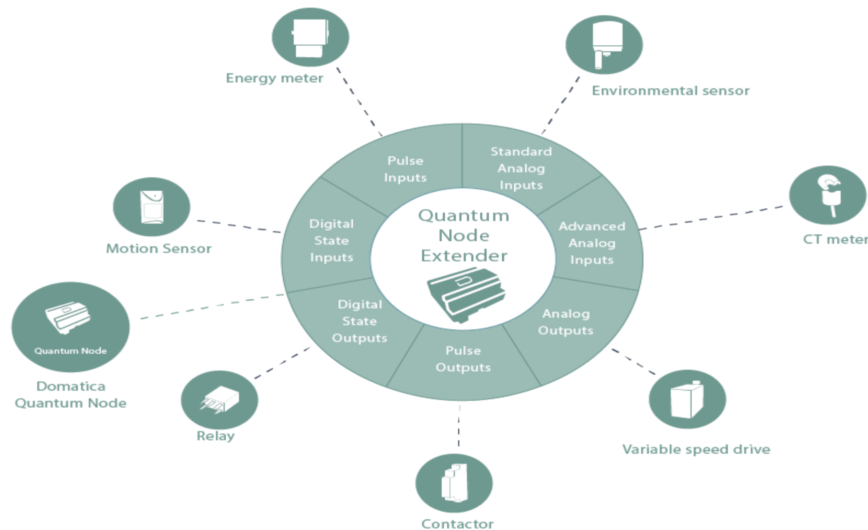

Figura 12 - Quantum Node Extender

(SDK) que permite o desenvolvimento de aplicações de *software* para várias plataformas e em várias linguagens de programação, como por exemplo C#, Java e C++. Este SDK possibilita que qualquer solução desenvolvida por terceiros possa comunicar com qualquer DQN permitindo o envio e a receção de dados, monitorização e controlo em real time. Também através do conceito de REST - Representational State Transfer -, é possível fazer esse tipo de operações com a integração deste mesmo SDK, reforçando uma vez mais a ideia da facilidade de integração e comunicação com os DQN.

Em suma, o DQN é adequado para Automação Industrial, Sistemas de Gestão de Edifícios, Sistemas de Gestão de Energia, Sistemas de Controlo Distribuídos, Smart Homes, aplicações M2M/IoT ou outras soluções de monitorização e controlo.

4.2 Domatica Quantum Cloud (DQC)

A Domatica Quantum Cloud (DQC) é um conector IoT que possibilita a interconexão entre os serviços disponibilizados na *cloud* e os dispositivos instalados no campo, como é o caso dos DQN, ou também de outros dispositivos não proprietários da Domatica. Esta interconexão é caracterizada por um vasto conjunto de serviços e protocolos.

O objetivo principal do projeto passou pelo desenvolvimento de uma Core API. Esta Core API caracteriza-se por ser independente do formato usado para a representação de Web API's. Esta API tem duas funcionalidades principais. A primeira era de permitir uma comunicação direta e em real time com os dispositivos instalados em campo, independentemente da sua rede ou localização. Por outro lado, a segunda funcionalidade principal é a disponibilização de funcionalidades e serviços, como por exemplo um conjunto de operações sob a base de dados (registo de utilizadores, consulta de informações, etc).

Esta Core API responde a pedidos REST, em formato JSON, e deste modo a comunicação entre o cliente/utilizador e a DQC, onde se encontra integrada a Core API, não é feita de forma totalmente direta. Para isso é necessário a criação de camada superior que permita essa mesma comunicação. Na próxima secção - 4.2.1 - Comunicação -, será explicado mais detalhadamente o funcionamento dessa comunicação.

Atualmente a arquitetura da DQC esta dividida em camadas, destacando-se essencialmente a camada L0 e a camada L1. A camada L0 é a camada base, responsável pela receção dos pedidos e da validação e processamento/execução dos mesmos. É nesta camada que se encontra atualmente integrada a Core API. Esta camada L0 é única e totalmente genérica, ou seja, é nela que se encontram desenvolvidas todas as funcionalidades, serviços, protocolos que servem de base para as outras camadas superiores. Por outro lado, a camada L1 é uma camada menos genérica em que o seu principal objetivo passa pelo envio de pedidos à camada L0 e pela receção da resposta desses mesmos pedidos. O desenvolvimento e funcionamento desta camada deve ter em consideração as funcionalidades que a camada L0 suporta. Podem ser desenvolvidas uma ou mais camadas L1, podendo cada uma delas suportar apenas determinadas funcionalidades da camada L0. Na Figura 13 - Domatica Quantum Cloud -, é possível ilustrar com maior detalhe o anteriormente explicado. Nela está representado a Domatica

Cloud Services (DCS) que corresponde a um conjunto de serviços que a Cloud disponibiliza. É importante referir que atualmente a DCS passou a ser tratada como DQC. Ao centro temos a camada do CORE (camada L0). A elas estão e/ou futuramente irão estar conectadas várias camadas L1. Camada umas destas camadas, como por exemplo a Domatica Portal, são camadas que estão interligadas à camada principal, tendo assim acesso a determinadas funcionalidades da camada L0, dependendo das funcionalidades a que determinada camada L1 foi desenvolvida para suportar.

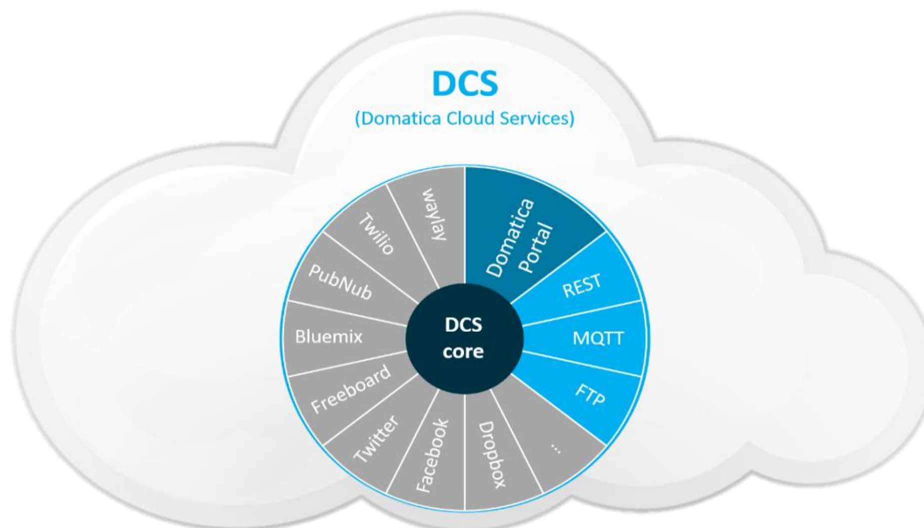

Figura 13 - Domatica Quantum Cloud

4.2.1 Comunicação

Tal como já foi referido anteriormente a DQC é caracterizada essencialmente por duas camadas. A camada L0 que é única e que representa a DQC - Core e a(s) camada(s) L1 que têm como objetivo o de estabelecer comunicação com a camada L0. Deste modo é sempre necessário a existência de pelo menos uma camada L1 para estabelecer qualquer tipo de comunicação com a camada L0 e ter assim acesso às funcionalidades que esta dispõe. É importante relembrar uma vez mais que a camada L0 responde a pedidos REST efetuados pela(s) camada(s) L1. REST é um protocolo de comunicação, considerado uma abstração da arquitetura World Wide Web (WEB) que explora essencialmente a tecnologia e protocolos existentes na WEB. Uma das suas principais vantagens é flexibilidade, uma vez que não existem restrições no formato da mensagem, podendo-se optar pelo formato mais adequado, sendo os mais comuns o JSON e/ou XML. O principal nesta arquitetura

REST são os Uniform Resource Locator (URL), fazendo uso dos principais métodos HTTP atualmente existente, como é exemplo o GET, POST, DELETE ou PUT. Consoante o URL utilizado bem como os dados com eles enviados, é possível aceder e executar diferentes tipos de pedidos, cada um deles correspondente a uma determinada funcionalidade. Na Figura 14 - REST API -, é ilustrado com maior detalhe o processo do funcionamento de um determinado pedido a uma REST API.

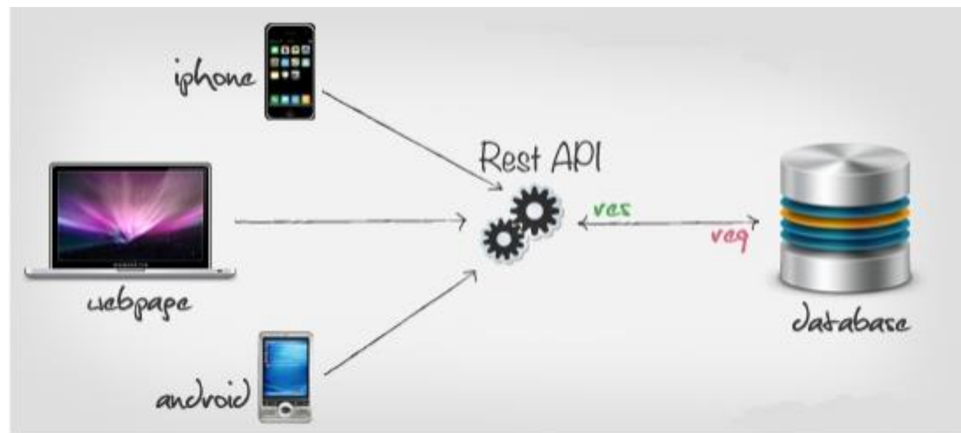

Figura 14 - REST API

Como é possível observar, utilizando diversos dispositivos, cada um deles com um diferente sistema operativo (a título de exemplo, Android, iOS e Windows) é possível efetuar um determinado pedido à API. Isto é possível uma vez que esta atua sobre os protocolos HTTP e deste modo é abstrata e genérica, permitindo uma maior flexibilidade com diferentes tipos de ecossistemas uma vez que cada um deles suporta qualquer um desses protocolos HTTP. Neste exemplo a REST API encontra-se conectada a uma base de dados onde é possível recolher a informação pedida pelo utilizador e entregar-lhe de volta. No caso mais concreto deste projeto, existe uma REST API que por sua vez se encontra conectada à CORE API. Esta recebe os pedidos pela REST API, encontrando-se também ligada a uma base de dados, em MySQL, e também interligada a todos a diversos dispositivos IoT inteligentes como é o caso dos DQN, ou de outros dispositivos não proprietários da Domatica, como por exemplo um Raspberry pi, permitindo assim executar um conjunto de tarefas sob os mesmos. Tanto o JSON como o XML são dois dos principais formatos muito utilizados hoje em dia em RESTful API's para a estruturação das mensagens. Apesar de ser possível implementar qualquer um dos formatos, é necessário ter em conta algumas especificações e características para a escolha recair sob o formato mais adequado. Atualmente não existe qualquer regra que defina qual dos

formatos é relativamente mais vantajoso, mais rápido ou mais simples. Consoante o tipo de solução a desenvolver, alguns dos formatos de mensagens poderão nesses casos apresentar mais vantagens em relação a outros. Desde modo é bastante importante ter conhecimento do tipo de solução a implementar, que tipos de dados são necessários e qual a sua extensão [30]. Na comunicação entre a camada L0 e a camada(s) L1 foi escolhido o formato de mensagem em JSON. Esta escolha deveu-se principalmente à pouca complexidade dos dados a transferir e por este apresentar uma maior legibilidade em relação a outros formatos de mensagem, como é o caso do XML. Uma vez que a velocidade de comunicação é um dos fatores mais importantes a ter em conta no funcionamento da API, o JSON oferecia assim uma maior vantagem uma vez que as operações de tratamentos dos dados neste mesmo formato apresentavam uma menor complexidade durante a sua implementação e processamento. Por outro lado, algumas das plataformas de IoT usadas pelos clientes da Domatica, como é o caso do Freeboard (https://freeboard.io), usam o JSON como um dos formatos principais, não havendo suporte por exemplo para formato XML.

4.2.2 Protocolo de Comunicação

Para a implementação desta solução era necessário recorrer a um *Message Broker* que desempenhasse um papel na comunicação entre as diferentes camadas. Um *Message Broker* - Figura 15 - Message Brocker - é essencialmente um programa intermediário, que tem como objetivo a tradução das mensagens do protocolo do remetente para o protocolo das mensagens do recetor, sendo assim responsável pela troca de mensagens entre diferentes aplicações [31].

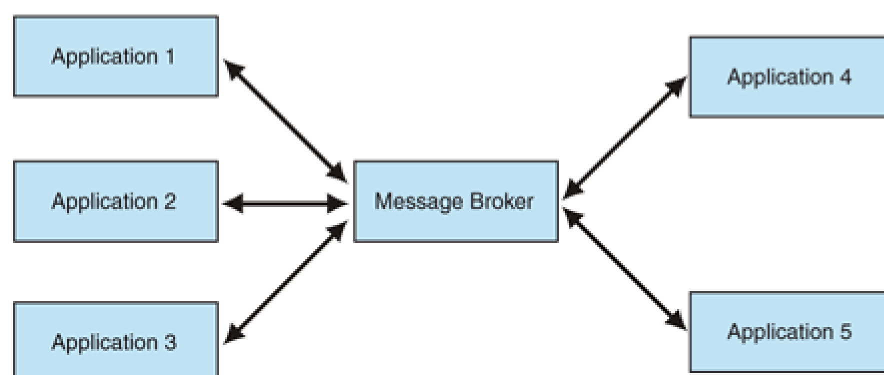

Figura 15 - Message Brocker

Por outro lado, é necessário escolher qual o protocolo de comunicação a implementar no *message broker*. Tendo em conta o aumento do número de dispositivos ligados à internet e a crescente comunicação entre os mesmos no mundo da *IoT*, a escolha do protocolo de comunicação é bastante importante uma vez que a comunicação entre os diversos dispositivos, têm como principal característica a rapidez e a fiabilidade. Os protocolos já existentes na Internet são considerados bastantes pesados para que a sua implementação fosse possível nas emergentes aplicações *IoT* que foram surgindo. Outra característica a ter em atenção, é que grande parque dos dispositivos *IoT* são de baixo consumo energético e com recursos bastante restritos. Deste modo houve a necessidade da implementação de protocolos que se caracterizassem pela sua leveza e que fossem passíveis de acompanhar os baixos recursos apresentados pelos dispositivos *IoT*.

Atualmente já existem definidos alguns protocolos de comunicação. Apesar de existirem mais protocolos, no mundo da *IoT* destacam-se principalmente o Advanced Message Queuing Protocol (AMQP), o Message Queuing Telemetry Transport (MQTT) e o Constrained Application Protocol (CoAP) [32].

O AMQP é um protocolo de comunicação em rede (tal como é o HTTP), permitindo a comunicação entre aplicações. Uma vez que existem diversos sistemas/aplicações com diferentes arquiteturas e linguagens de programação, este protocolo vem tornar a integração destes mais simples e também permitir que esses diferentes sistemas/aplicações possam comunicar entre si [33].

As suas duas principais vantagens são a confiabilidade e interoperabilidade entre diversas tecnologias e plataformas. Neste protocolo existem dois principais intervenientes: Os consumidores e os produtores. Os produtores são responsáveis por publicar as mensagens e enviar as mesmas até às *exchanges*. As *exchanges* distribuem as mensagens para as filas onde posteriormente são entregues aos consumidores [34]. Na

Figura 16 - Arquitetura AMQP é possível observar a arquitetura deste protocolo.

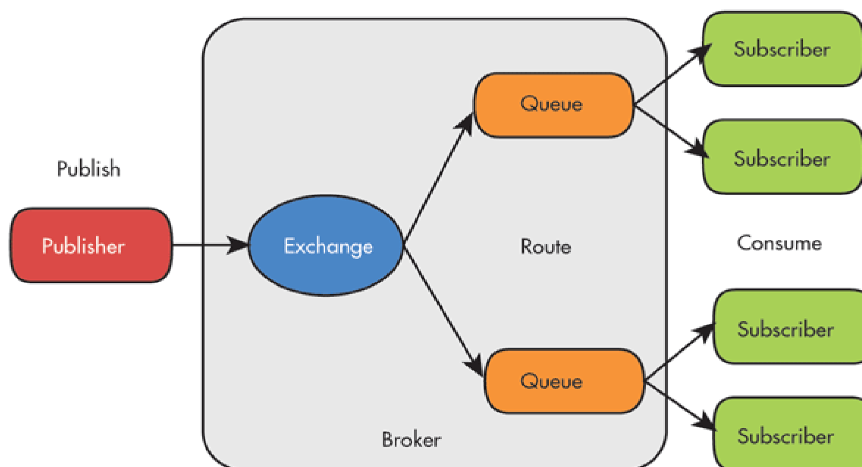

Figura 16 - Arquitetura AMQP

Outras das vantagens do AMQP é o facto de este ter a implementação de *acknowledgments* de mensagens, o que permite que quando uma mensagem é enviada a um consumidor, este pode-a devolver de volta o *acknowledgment*, dando informação ao produtor que a mesma foi entregue. Quando uma mensagem, por algum motivo, não é entregue ao consumidor, o AMQP permite que sejam configuradas algumas regras ao nível do *broker*, para definir qual o comportamento a implementar nesses casos. A mensagem não entregue, pode ser por exemplo descartada, ser retornada de volta ao seu produtor ou colocada numa fila de espera [33].

Por outro lado, o MQTT, desenvolvido pela IBM e pela Eurotech, é baseado em TCP/IP. Este protocolo já existia há algum tempo (criado em 1999), sendo recentemente adaptado à realidade exigida pelos dispositivos IoT. Apesar de se encontrar na mesma camada que o protocolo HTTP, o *playload* usado no HTTP é bastante maior ao que é usado no MQTT, o que não permitiria o seu uso em ligações de baixa qualidade. Este protocolo, apesar de ter algumas semelhanças com o AMQP, é menos sofisticado (não existem filas de espera por exemplo) e a sua implementação é mais simples, sendo projetado principalmente para dispositivos com baixos recursos de largura de banda [35].

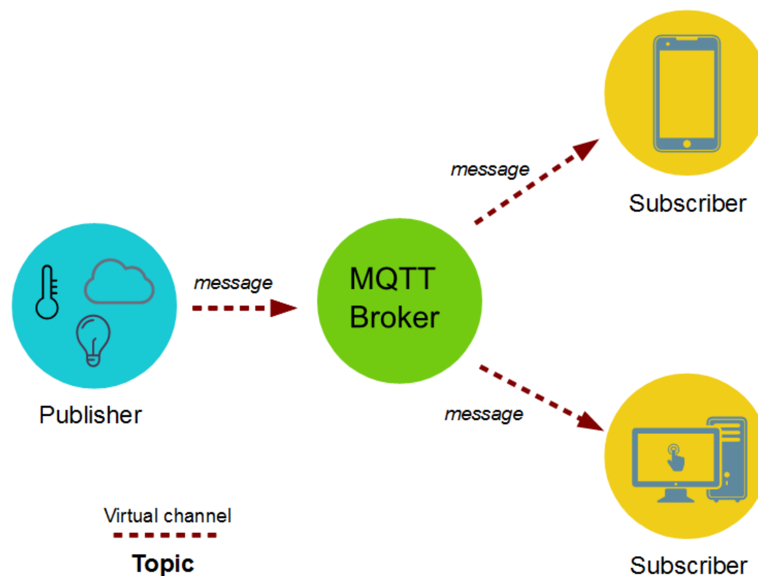

Figura 17 - Arquitetura MQTT

Este protocolo utiliza um padrão de mensagens *publish/subscribe* e uma vez mais à semelhança do AMQP, o *broker* tem o papel de intermediário na troca de mensagens. Neste processo de troca de mensagens é possível configurar níveis de qualidade de serviço, vulgarmente conhecidos como Quality of Service (QoS). Existem três níveis (QoS 0, QoS 1 e QoS 2), que definem essencialmente o comportamento entre os elementos que estão envolvidos no processo de comunicação. No nível de QoS 0 por exemplo, não há garantia da entrega da mensagem nem qualquer tipo de informação se a mesma foi efetivamente entregue. No nível de QoS 1 existe a confirmação de que a mensagem foi realmente entregue e a mesma é armazenada por quem envia, até receber informação de que foi entregue. No entanto a mesma mensagem pode ser entregue várias vezes caso haja algum *delay* no processo de envio da mensagem. Em QoS 2 há garantia da entrega da mensagem apenas uma única vez, havendo mensagens de confirmação nos dois sentidos, tanto do lado do publisher como do lado do *subscriber*. É importante referir que cada nível de QoS é negociado entre o cliente e o *broker*. Isto irá permitir que uma certa mensagem possa ter níveis de QoS diferentes entre *o publisher – broker* e entre o *broker – subscriber* [35].

O MQTT foi desenhado essencialmente para telemetria[2] remota de aplicações, onde foram definidos três objetivos de desenho específicos [36]:

[2] Refere-se geralmente a comunicações sem fios

- Deve oferecer uma entrega segura e única, de modo a permitir que uma mensagem seja transferida de forma confiável, entre um sensor remoto e a solução implementada em *back-end*;
- Este protocolo deve ser o mais leve possível, uma vez que o mesmo é integrado em redes com baixa largura de banda, minimizando assim a sobrecarga em cada mensagem;
- Deve ser extremamente fácil a sua implementação nos diversos dispositivos e sensores utilizado nas variadas soluções IoT.

Como já foi referido, este protocolo segue o um paradigma de *publish/subscribe.* Deste modo esse paradigma pode ser decomposto tendo em conta as seguintes três dimensões [37]:

- Desacoplamento espacial: as partes envolvidas durante a troca de mensagens não necessitam de se conhecer. Quem envia as mensagens (*publisher*) não precisa de saber a localização de quem as vai receber (*subscriber*), nem quantos deles vão receber. Da mesma forma, quem recebe as mensagens não têm qualquer tipo de referência sobre quem as enviou.
- Desacoplamento temporal: as partes que comunicam, não precisam de manter ativamente interação no mesmo espaço temporal. Deste modo, um agente pode publicar (*publisher*) uma mensagem sem que o agente recetor (*subscriber*) esteja por exemplo conectado naquele preciso momento.
- Desacoplamento de sincronização: os *publishers* não são bloqueados enquanto se encontram a produzir qualquer tipo de evento, bem como os *subscribers* que podem, de forma assíncrona, receber notificações sobre a ocorrência de um evento enquanto realizam outro tipo de atividade concorrente.

Por sua vez o CoAP é um protocolo cliente/servidor que foi criado desde início tendo em conta exclusivamente dispositivos IoT, ao contrário do que aconteceu com o MQTT, como já foi referido anteriormente. Foi construído para operar com HTTP e *RESTful*, tornando-o totalmente compatível com a internet.

O CoAP opera sob o UDP, ao contrário do que acontece com o AMQP e o MQTT – que operam sob o TCP – o que torna as comunicações deste protocolo menos confiáveis. Um aspeto que este protocolo apresenta em relação aos seus opositores é o facto de este permitir requisitos em *multicast*. Isto é possível uma vez que foi construído com base no

IPv6, que permite endereçamento *multicast* nos dispositivos [38]. No entanto a entrega de mensagens em *multicast* poderá afetar por exemplo a vida útil da bateria do dispositivo. No modo de operação deste protocolo, as mensagens são enviadas e recebidas usando como base o modelo de *request/report* (pedido/relatório). As mensagens usam um cabeçalho base simples, em formato binário. Na Figura 18 - MQTT vs CoAP - é possível verificar as principais diferenças entre o MQTT e o CoAP, com especial atenção ao uso do *multicast* no CoAP.

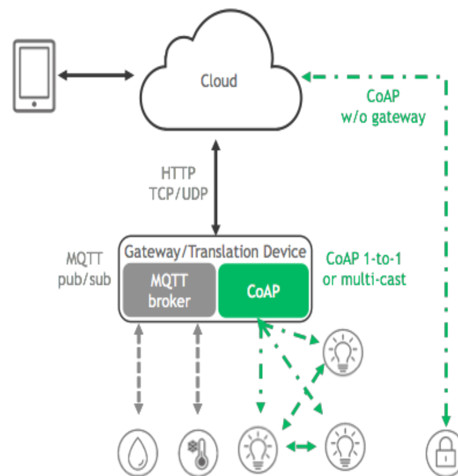

Figura 18 - MQTT vs CoAP

Após o estudo dos principais protocolos de comunicação, atualmente em uso no mundo da IoT, a escolha para esta solução recaiu sobre o AMQP, uma vez que à semelhança do MQTT é um dos mais usados. Existem algumas diferenças entre estes dois protocolos de comunicação, que é importante destacar. Um deles refere-se à segurança da conexão. O MQTT não tem implementado qualquer mecanismo de segurança da conexão. Por sua vez, o AMQP, integra TLS de formar a garantir maior segurança durante a conexão. Em termos de implementação, o MQTT poderá de certa forma ser mais fácil de implementar por ser um protocolo mais pequeno e leve. No entanto, tanto o MQTT como AMQP podem ser implementados em dispositivos com quantidades muito baixas de memória.

Escolhido o protocolo a usar, foi então necessário selecionar o *broker* a implementar para a troca de mensagens. Um dos principais critérios era que esse mesmo *broker* tivesse suporte para o protocolo escolhido - o AMQP. Durante algumas pesquisas relativamente aos protocolos anteriormente referidos, foi possível observar que um dos *brokers* mais referido era o RabbitMQ. Este *broker* é um software *open source* e foi originalmente criado para dar suporte ao AMQP. Atualmente dá suporte a outros protocolos, entres eles o

MQTT. É um dos *brokers open* source mais populares e é usado por grandes empresas, tais como Ford, Cisco, Instagram, entre outras. É leve e a sua implementação em *cloud* é bastante simples podendo ser configurado com requisitos de alta disponibilidade. Outras das grandes vantagens do RabbitMQ é sua capacidade de integração em diversos sistemas e com suporte para diversas linguagens de programação. Este último ponto foi bastante importante e decisivo na sua escolha, uma vez que era necessário a utilização de um *broker* que além de ter suporte para o protocolo AMQP, deveria também ter suporte para diversas linguagens de programação, entre elas o Java e o C++, uma vez que era necessário estabelecer a troca de mensagens entre a camada L0 (C++) e a(s) camada(s) L1 (Java). Na figura Figura 19 - RabbitMQ, é possível observar o sistema deste *broker*, sendo esse sistema semelhante ao descrito para o protocolo AMQP.

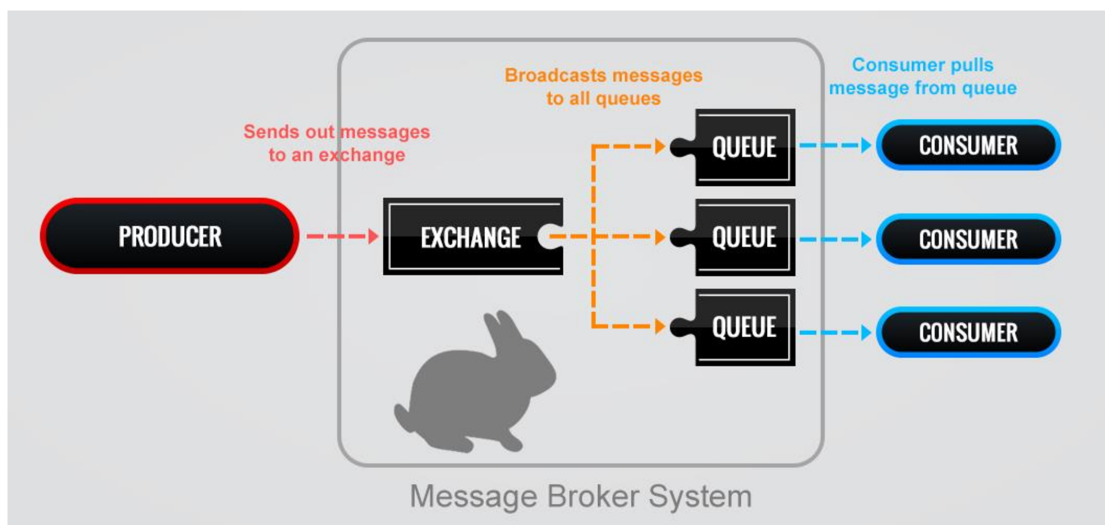

Figura 19 - RabbitMQ

Para a configuração e manutenção do servidor RabbitMQ, a interface gráfica – *rabbitmq-management plugin* -, podendo a mesma a ser acedida através do *browser*, através do endereço e porta do servidor. Este *plugin* inclui um conjunto de recursos:

- Declara, lista e remove filas de espera, ligações e utilizadores;
- Monitoriza o tamanho das filas de espera, a taxa de mensagens por fila, bem como a taxa de conexão de dados;
- Envia e recebe mensagens;
- Possibilidade de importar e exportar todas as configurações em formato JSON;

- Possibilidade de forçar o fecho de conexões bem como a limpeza das filas de espera.

Deste modo, tanto a camada L0 como a L1, quando inicializadas, são conectadas automaticamente ao servidor de RabbitMQ, encontrando-se este servidor configurado e a ser executado numa máquina com o sistema operativo Linux. Isto é possível, uma vez que ambas as camadas, possuem um ficheiro de configurações. Neste ficheiro de configurações, está definido o endereço IP e porta do servidor, um utilizador previamente criado no servidor RabbitMQ bem como o nome das filas de espera necessárias. Neste projeto foram definidas duas filas de espera. Uma para o envio das mensagens e outra para a sua receção.

4.2.3 Camada L0

Como já foi referido a camada L0 é a camada base e principal da DQC. É esta camada responsável pela receção de todos os pedidos provenientes da(s) camada(s) L1, processamento desses mesmos pedidos, a sua validação, bem como a construção da resposta a ser entregue à camada L1 que efetuou o respetivo pedido, de modo a que esta camada L1 entregue a resposta de volta ao utilizador. A velocidade de processamento de todos os pedidos, uma vez que poderão ser efetuados ao mesmo tempo milhares de pedidos, é um dos fatores que mais importância foi dada. Era essencial a implementação dos diversos requisitos necessários, mas a velocidade e a fiabilidade dos mesmos era dos fatores imprescindível para a sua implementação.

Tal como já foi anteriormente referido, esta camada tem atualmente duas principais funcionalidades. A primeira era a da integração do SDK o que iria permitir a comunicação com os DQN e com outros dispositivos IoT inteligentes, para a leitura e processamento de dados, bem como de outro tipo de funcionalidades que serão alvo de uma detalhada explicação posteriormente. Por outro lado, a segunda principal funcionalidade era a implementação de um conjunto de funcionalidades que operassem sobre uma base de dados MySQL. Isto permitia por exemplo a criação de novos utilizadores, o registo dos vários dispositivos IoT, obter informações dos vários utilizadores, ou guardar o histórico (*logs*) dos eventos gerados por cada dispositivo, entre outras operações. É expectável a existência de milhares de pedidos simultaneamente e como tal a API devia de igual modo responder a esses mesmos pedidos no menor tempo possível. Um bom exemplo é por exemplo a leitura instantânea de um sensor de temperatura. Algumas das plataformas IoT

usadas tanto pela Domatica, mas principalmente pelos seus clientes, é o Freeboard. O Freeboard é uma plataforma que permite a configuração de diversas fontes de dados. Essas mesmas fontes de dados, poderão ser por exemplo o formato JSON. Assim é possível a configuração de um ou vários pedidos REST a um determinado dispositivo IoT e assim obter o valor instantâneo da temperatura naquele exato momento. Este pedido pode ser configurável para ser executado a todos os segundos, sendo o seu valor mostrado na Dashboard do Freeboard. Deste modo, um único cliente poderá estar a cada segundo a efetuar vários pedidos simultaneamente a um dispositivo, podendo haver também outros clientes a efetuarem o mesmo ou até outro pedido simultaneamente ao mesmo dispositivo, noutro determinado sensor (CO_2, humidade, etc.). Como é possível observar o processamento destes dados têm de ser efetuados rapidamente, na ordem dos milissegundos, uma vez que não fazia sentido no exemplo anterior a leitura da temperatura demorar mais que milissegundos, deixando assim essa leitura de ser realista uma vez que o valor apresentado não corresponderia ao valor real naquele exato momento. Por outro lado, se por exemplo, usarmos um determinado dispositivo para o controlo de luzes, é expectável que quando executamos um determinado evento (ligar/desligar) que esse mesmo evento produza resultados instantâneos. Esta velocidade de processamento depende essencialmente de alguns fatores. Um deles é do SDK. Para a comunicação com os diversos dispositivos IoT, independentemente do ecossistema (sistema operativo, linguagem de programação, etc.) em que nos encontremos, é necessário integrar o SDK fornecido pela Domatica. Este tem de ser construído com o maior rigor de modo a diminuir ao máximo a possível latência existente para a execução e leitura de eventos nos dispositivos. No caso mais concreto desde projeto, já era expectável a existência de alguma latência uma vez que qualquer pedido efetuado operava sob o protocolo HTTP, e não localmente como poderia ser expectável numa solução software mais convencional. Por outro lado, todos os pedidos recebidos na *cloud* tinham de ser processados previamente antes de serem executados. Este processamento inicial dependia de diversos fatores, entre eles a leitura das mensagens em JSON e sua interpretação, validação e recolha de dados na base de dados. Deste modo, a linguagem de programação C++ foi a linguagem principal escolhida para a implementação da solução na camada L0. Existem essencialmente dois tipos de linguagens de programação. As de alto nível e as de baixo nível. As de alto nível caracterizam-se essencialmente por serem de rápida aprendizagem uma vez que o nível de abstração é elevado. São mais distantes da linguagem máquina e assim sendo, mais

próximas da linguagem humana. Exemplos deste tipo de linguagens são por exemplo o C#, Java ou PHP [39].

Por sua vez, uma linguagem de baixo nível, é mais próxima do processador. Isto quer dizer que compreende as características arquiteturais do computador, utilizando assim as instruções do processador. A sua principal vantagem é a velocidade para compilar e processar os dados. Uma linguagem considera de baixo nível é por exemplo Assembly. Estas linguagens caracterizam-se pela sua baixa portabilidade uma vez que um dado código executado numa máquina é impossível de ser executado numa outra máquina [39].

Atualmente há quem considere C/C++ linguagens de alto-nível ou de baixo nível. Isto deve-se ao facto de que C/C++ são as linguagens mais próximas a uma linguagem de baixo nível, como do Assembly, até porque necessitam de utilizar compiladores ao contrário das linguagens de alto nível. O C/C++ representam uma variedade de linguagens, uma vez que a maioria das linguagens existentes utilizou C/C++ para a construção das suas próprias bibliotecas. Por estes fatores anteriormente descritos pode-se considerar ambas estas linguagens simultaneamente como alto nível e baixo nível [39].

O C++ é usado essencialmente em aplicações que necessitam de processar grandes quantidades de dados em pouco tempo. Este era um fator primordial uma vez que esta camada necessitava de processar grandes quantidades de dados no menor tempo possível. Por outro lado, estes tipos de linguagens, e considerando neste caso específico o C++ como de baixo nível, apresentam algumas desvantagens. Uma delas é a dificuldade de aprendizagem sendo necessário uma maior experiência e conhecimento para que o seu uso e implementação tenha a eficiência esperada e mais correta.

Uma linguagem como o C++ obriga a que seja feita uma gestão manual por parte do programador da memória interna a ser usada. Em linguagens como o Java, por exemplo, essa gestão é feita de forma automática. Esta gestão manual da memória significa que cada vez que é alocada memória é necessário libertar a mesma, quando esta já não esteja em uso e ou que não seja mais necessária. Por se tratar de uma gestão manual, ou seja, da responsabilidade de quem está a construir a solução - o humano, a probabilidade de haverem falhas e consequentemente fugas de memórias torna-se muito mais elevada.

Este facto foi um dos grandes desafios encontrados no desenvolvimento desta camada. Sendo o CORE, uma API que está sempre em execução na *Cloud*, a gestão da memória alocada, essencial para a implementação das variadas funcionalidades, era sem

dúvida alguma um dos pontos mais sensíveis e importantes. Caso houvesse uma incorreta gestão da memória alocada pelos diversos objetos e classes, poderia haver o risco da existência de *memory leaks*. Um *memory leak* acontece quando parte da memória alocada não é libertada quando a mesma já não se encontra em uso. Esta ocorrência pode levar a falhas do sistema, quando a memória do mesmo estiver completamente consumida [40].

Neste caso em concreto, caso existissem ocorrências de *memory leaks*, os mesmos e com a constante utilização da API, iriam originar a uma falha no sistema e consequentemente na API. Esta falha, resultante de consecutivos *memory leaks* iriam originar a que esta API deixasse de funcionar e de estar acessível a todos os utilizadores.

Esta situação iria originar alguns problemas graves, quer a nível interno como a nível externo. A nível interno porque a empresa iria ver uma solução desenvolvida por si com risco de deixar de funcionar a qualquer momento. Por outro lado, a nível externo, uma vez que os clientes iriam ser prejudicados. Neste caso os clientes ficavam impossibilitados de puder fazer pedidos à *cloud* e consequentemente todas as suas próprias soluções que dependeriam desta API também não iriam estar funcionais.

Haviam algumas situações para contornar este problema. Uma delas e sem dúvida alguma a mais importante, era o conhecimento. Se o programador tiver um bom conhecimento da linguagem, bem como o seu funcionamento, consegue na esmagadora maioria das vezes, desenvolver o seu código para que este faça a gestão correta da memória. Por outro lado, existem já algumas ferramentas, como é o caso do Valgrind, que conseguem detetar fugas de memórias nas aplicações. Este tipo de ferramentas são bastantes importantes e devem ser usadas sempre que possível na fase de testes da aplicação, garantindo assim a inexistência de possíveis fugas de memória. O uso desta ferramenta e seu funcionamento será posteriormente explicado no capítulo 5.2 - Testes.

As funcionalidades implementadas nesta camada serão explicadas detalhadamente no capítulo 5.1 – Funcionalidades implementadas.

4.2.4 Camada L1

Como já foi referido, o papel principal da camada L1 é o de estabelecer comunicação com a camada L0 para o envio e receção de dados.

Isto deve-se essencialmente a dois fatores. Um deles, a nível da segurança. Com a implementação de camadas superiores – camadas L1 - à camada L0, era implementado um

maior nível segurança, uma vez que todos os pedidos efetuados são inicialmente processados pela camada L1, não havendo qualquer tipo de pedido direto à camada L0. Isto é um fator importante uma vez que a camada L0 é uma camada mais sensível, uma vez que a ela estão conectados todos os dispositivos IoT, bem como a base de dados.

Por outro lado, outro fator é o nível organizacional. A camada L0, é genérica e nela estão implementas todas as funcionalidades, protocolos e serviços necessários. No entanto nem todas as camadas L1 acedem ou necessitam de aceder a todas essas mesmas funcionalidades. Deste modo existe a necessidade do desenvolvimento de camadas mais específicas - L1 -, sendo cada uma delas desenvolvida para corresponder a funcionalidades específicas da camada L0, como por exemplo a camada L1 – *System* Portal, desenvolvida também durante este projeto. Esta camada tinha como objetivo o de fornecer suporte à página web - Quantum Portal -, desenvolvida pela outra equipa de desenvolvimento. Esta camada foi implementada com base nas funcionalidades oferecidas pela camada L0, mas dando apenas suporte a uma parte dessas funcionalidades, funcionalidades mais específicas ao Quantum Portal. Exemplo disso são as funcionalidades de listagem de utilizadores, de registo de utilizadores, registo de dispositivos IoT. Por outro lado, nada impede que seja implementada outra camada L1, a usar numa outra solução, e que sirva também ela de suporte a funcionalidades já existentes no L1 – *System* Portal. Este facto torna a atual solução mais robusta e organizada uma vez que podemos ter várias camadas L1 específicas para diferentes tipos de soluções, ao invés de termos uma camada L1 genérica que servisse de suporte a todos os tipos de soluções desenvolvidas. É importante referir que futuramente irão ser implementados novos protocolos e serviços, como é exemplo do FTP, MQTT, Dropbox, Twitter, Facebook, entre outros. Cada um deles é representativo de uma camada L1 independente e específica para esse determinado protocolo ou serviço. Para o desenvolvimento da camada L1, mais propriamente a L1 – *System* Portal, foi escolhida a linguagem de programação Java. Esta escolha deveu-se principalmente pelo facto de nesta camada não existir processamento de grandes quantidades de dados, uma vez que a sua responsabilidade é apenas o de estabelecer conexão com a camada L0 - através do *broker* RabbitMQ -, bem como o de enviar e receber os dados no formato JSON. Deste modo, a utilização de uma linguagem de alto nível, como é o caso do Java ajustava-se perfeitamente para a implementação desta camada, uma vez que a tarefa da codificação dos dados em formato JSON era relativamente rápida e simples de implementar. Para a codificação dos dados em JSON, foi utilizada a biblioteca GSON. O GSON foi criado pela

Google e tem como objetivo a serialização e deserialização de objetos Java para formato JSON e vice-versa. Se tivermos como exemplo a Figura 20 - *Class Person*, podemos observar que a *Class Person* tem dois atributos: *name* e *age*. Utilizando o Gson estes objetos Java serão codificados em formato JSON – Figura 20 - *Class Person*.

```
Public class Person
{
    public String name;
    public int age;

    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

Figura 20 - Class Person

```
{
    "name": "person name",
    "age": 24
}
```

Figura 21 - Formato JSON

Uma das principais vantagens de utilização do Gson é a facilidade de transformar um objeto Java em formato JSON e vice-versa. Por outro lado, ao utilizarmos a biblioteca Gson ao nível da camada L1, conseguimos garantir que num determinado pedido, os dados em JSON a enviar para a camada L0 correspondem exatamente aos objetos previamente definidos. Isto quer dizer que nesse pedido, em que seja necessário enviar em JSON por exemplo o campo "name" e "age", o Gson garante a validação desses campos, não aceitando outro atributo, por exemplo - "namee" ou "agee", etc. Isto é um pormenor importante, primeiro porque garante uma mais correta estruturação dos dados a enviar e segundo garante que só seja efetuado um pedido à camada L0 se os dados corresponderem ao que foi previamente definido. Caso não correspondam, o pedido não é efetuado e a camada L1 constrói uma mensagem de erro a devolver de volta ao utilizador.

5. Implementação, Testes e Segurança

Para um maior conhecimento da solução desenvolvida é importante descrever e explicar todas as funcionalidades implementadas. Nesta secção serão abordadas individualmente cada uma das funcionalidades desenvolvidas, onde será feita uma descrição geral de cada uma delas.

Por outro lado, para garantir o correto funcionamento da solução, garantindo que a mesma corresponda aos requisitos previamente definidos, foi necessário a realização de alguns tipos diferentes de testes. Estes testes, bem como algumas ferramentas que auxiliaram no seu desenvolvimento, encontram-se descritas no capítulo 5.2 Testes e Ferramentas.

Também a segurança é sempre um aspeto a ter em consideração durante o desenvolvimento de uma solução, tenho em conta o crescimento exponencialmente dos dispositivos IoT. Todos os detalhes implementados e tidos em conta, são abordados no capítulo 5.3 Segurança.

5.1 Funcionalidades Implementadas

Neste capítulo serão descritas individualmente todas as funcionalidades implementadas. No entanto é importante referir que no Capítulo – Anexos é possível consultar toda a documentação oficial da Core API, desenvolvida em paralelo ao longo do projeto, servindo de suporte às funcionalidades descritas de seguida. Nesta documentação encontram-se descriminadas detalhadamente todas as funcionalidades, o seu funcionamento bem como a formatação em JSON necessária para cada uma delas.

5.1.1 Métodos de autenticação

Para os diversos pedidos a serem feitos à CORE API era necessário garantir que de alguma forma esses pedidos fossem efetuados por utilizadores com autorização para tal, garantindo assim que esses mesmos pedidos não eram efetuados por qualquer outro utilizador. Desde modo foram implementados 3 métodos de autenticação distintos. Estes métodos de autenticação eram sempre obrigatórios em cada pedido, podendo existir funcionalidades que suportem qualquer um deles, como outras que suportem apenas um dos métodos de autenticação. É importante referir que certas funcionalidades estão apenas

disponíveis para certos tipos de utilizador (por exemplo administrador), e deste modo durante o processo de autenticação é feita a verificação entre o tipo de utilizador que se esta autenticar com o tipo de utilizador permitido para aquela funcionalidade.

- **User Authentication**
  - Este método de autenticação utiliza como recurso o email e password do utilizador

  Request body:

```
{
   "authenticationMethod": {
      "userAuthentication": {
         "userEmail": "user@mail.com",
         "userPassword": "userPassword"
      }
   }
}
```

- **SessionId**
  - Este método de autenticação utiliza como recurso um sessionId. Este sessionId era gerado por um algoritmo consoante o *email* e *password* do utilizador bem como a hora exata em que essa criação era feita. Tinha um tempo de expiração de 30 minutos após a sua criação e o mesmo era renovado por mais 30 minutos cada vez que era usado;
  - Para a criação de um sessionId era necessário efetuar um pedido à Core API, onde o *body* desse pedido era o mesmo usado do User Authentication. Após esse processo, era criado um sessionId automaticamente, enviando o mesmo de volta ao utilizador.

  Request body:

```
{
  "authenticationMethod": {
```

```
            "sessionAuthentication": {
                "sessionId":"8DBNVX7387RA7ZNSM781"
            }
        }
    }
```

- **Token authentication**
    - Este método de autenticação utiliza como recurso um *token*. Este *token* era criado tendo como base o userId de um utilizador e o vNodeId de uma *gateway*. O vNodeId é um id virtual que é gerado automaticamente durante o processo de registo de uma *gateway*.

      Request body:

```
{
  "authenticationMethod": {
        "tokenAuthentication": {
              "gToken":"AABBCCDDEEFF990011223
        34455667788"
              "vNodeId":12345678912345678123
        }
  }
}
```

5.1.2 Gestão de utilizadores

A CORE API foi concebida tendo em conta a gestão de utilizadores. Deste modo foram desenvolvidas algumas funcionalidades que permitissem efetuar algumas operações ao nível dos utilizadores. A um utilizador está sempre associada uma entidade (ou empresa). Deste modo, quando ainda não existe uma entidade registada é necessário criar um registo. Este registo compreende dois tipos diferentes de informação. A primeira informação corresponde à informação de uma entidade, como por exemplo o nome, morada, website, setor de atividade, número fiscal, email, entre outras. A segunda informação corresponde à informação do utilizador a associar à entidade (nome, email, id,

password, etc.). Após este processo de registo, este mesmo utilizador pode adicionar novos utilizadores à sua entidade através da funcionalidade – New User.

- **New User**

Esta funcionalidade permite que sejam adicionados novos utilizadores a uma entidade. Neste processo são registadas informações pessoais do utilizador (id, nome, email, password, etc) que podem posteriormente ser editadas pelo menos através da funcionalidade Edit User. Esta operação é efetuada por utilizadores que tenham níveis de privilégios para tal (nível de *Owner* ou de administrador).

Request body:


```
{
  "entityInformation": {
       "entityId": "entity Id"
  },
  "userInformation": {
       "firstName": "user first name",
       "lastName": "user last name",
       "dateOfBirth": "02-19-1981",
       "telephone": 222222222,
       "userId": "userId1",
       "email": "userId1@mail.com",
       "userPassword": "pwd123#"
  }
}
```


- **Edit User**

Esta funcionalidade permite que um dado utilizador possa editar algumas das suas informações pessoais, como por exemplo a *password*, *email* ou número de telefone.

Request body:


```
{
  "userInformation": {
```

```
        "userId":"userId1",
        "newUserPassword": "newPwd123#",
        "newEmail": "usernewEmail@email.com",
        "newPhoneNumber": 9898989898
    }
}
```

- **User Information**

Esta funcionalidade permite que seja possível consultar as informações pessoais de um ou vários utilizadores simultaneamente, através dos seus userId's.

Request body:

```
{
    "entityInformation": {
        "entityId": "entityId"
    },
    "usersInformation": {
        "users": [{
             "userId": "userId1"
             },
             {
             "userId": "userId2"
             }]
    }
}
```

- **Delete User**

Esta funcionalidade permite que um ou vários utilizadores possam ser eliminados simultaneamente por outro utilizador, desde que esse tenha níveis de privilégios para executar essa operação.

Request body:

```
{
   "userInformation": {
        "users": [{
```

```
                "userId": "userId1"
            },
            {
                "userId": "userId2"
           }]
        }
        "entityInformation": {
            "entity": "entity id"
        }
    }
```

5.1.3 Gestão das Entidades

Como já foi referido qualquer utilizador tem de estar obrigatoriamente associado a uma entidade e por sua vez também os dispositivos IoT. Durante o processo de Register User, além do registo do utilizador base, era também registada uma entidade, ficando a mesma associada a esse utilizador. Posteriormente era possível efetuar algumas operações ao nível destas entidades.

- **Edit Entity**

  Esta operação tinha como objetivo editar algumas das informações definidas para uma determinada entidade durante o processo de registo. Estas informações a editar podem ser por exemplo o código postal, *website*, estado, cidade, número de telefone, entre outras.

  Request body:

```
{
 "entityInformation": {
   "entityId": "entityId",
   "vatID": 9069993232,
   "newCompanyName": "new company name",
   "newPhoneNumber": 111222333,
   "newAddress": "new entity address",
   "newState": "new entity state",
```

```
        "newZipCode": 6666,
        "newCountry": "new entity country",
        "newAccountContact": 99999,
        "newWebSite": "waw.newEntityWebsite.pptt"
        }
    }
```


- **Get Entity Info**

  Esta funcionalidade permitia que um utilizador, desde que estivesse associado à entidade em questão, pudesse consultar as informações das mesmas.

  Request body:


```
{
  "entityInformation": {
         "entityId": "entityId"
   }
}
```


5.1.4 Gestão das Gateways/Nodes IoT

As *gateways/nodes* IoT são os dispositivos inteligentes com as quais os diversos utilizadores podem comunicar. Estes dispositivos podem ser os dispositivos proprietários da Domatica ou outros tipos dispositivos não proprietários da Domatica, mas que estejam integrados com o *software* da empresa.

- **Claim node**

  Esta funcionalidade permite que uma *gateway/node* sejam registadas na CORE API. Quando um dispositivo é fabricado pela empresa ao mesmo é atribuído um *serial number* e uma *label key*. A *label key* é outro código que fica associado ao dispositivo de modo a conferir outro nível de segurança ao mesmo, não ficando este dependente apenas do *serial number*. No entanto, como já foi referido existem outros dispositivos não proprietários da empresa (por exemplo um Raspberry pi) que também estão integrados com as soluções da empresa. Nestes casos a estes dispositivos não está associado um *serial number* e *label key*. Deste

modo era necessário criar um mecanismo que de alguma forma fosse possível validar este tipo de dispositivos.

Assim no processo de registo, além da entidade a associar ao dispositivo, bem como algumas informações (latitude, longitude, nome, *timezone*, etc) é necessário definir qual o método de validação a usar – Node validation method.

Exemplo de um Node validation method (registration code)


```
{
    "nodeValidationMethod": {
          "registrationCodeMethod": {
              "regCode":"7HDADNAG3477ABDDAB92BKJDASHDA7
              3"
          }
     }
}
```


Este método de validação pode ser através do *serial number* e *label key*, onde é feita a verificação através de um algoritmo de modo a garantir que existe combinação certa entre o *serial number* e a *label key*. Caso seja um dispositivo não proprietário da Domatica é necessário recorrer a um *software* externo, que esteja instalado no dispositivo, para este gerar de forma aleatória um código de registo (*registration code*). Após este código ser gerado, o mesmo é guardado na base de dados da CORE API (com um determinado tempo de expiração) e assim passível de ser usado posteriormente pelo o utilizador, quando este efetua o processo de registo do seu dispositivo.

Quando o registo dos dispositivos é efetuado com sucesso, o mesmo devolve ao utilizador o *serial number* e *a label key*, uma vez que no caso de ter sido utilizado um dispositivo não proprietário, com recurso a um *registrarion code*, é gerado durante este processo um *serial number* e a uma *label key* a serem atribuídos ao dispositivo. Caso tenha sido registado um dispositivo proprietário é devolvido ao utilizador o mesmo *serial number* e *label key*. Também durante este processo é gerado um vNodeId (id virtual da *gateway*). Este vNodeId irá ser sempre usado em qualquer pedido que seja possível comunicar com o dispositivo, evitando

assim usar o *serial number* e *label key*. Este pormenor é importante uma vez que é conferido mais um nível de segurança, podendo cada utilizador (com níveis de privilégios mais baixos) comunicar com qualquer dispositivo da sua entidade sem nunca ter acesso ao *serial number* e *label key*.

Request body (com uso de registration code):


```
{
    "nodeValidationMethod": {
           "registrationCodeMethod": {
               "regCode":"7HDADNAG3477ABDDAB92BKJDASHDA7
               3"
           }
     },
    "entityInformation": {
         "entityId": "entity Id"
    },
    "nodeInformation": {
         "timeZone":   "(GMT)   Western   Europe   Time,
         London, Lisbon, Casablanca, Greenwich",
         "geoLat": 38.7,
         "geoLong": -9.14,
         "client": "Client",
         "tags": "Tags",
         "name": "Name"
    }
 }
```


- **Release node**

  Esta funcionalidade permite que um dado dispositivo seja apagado do sistema, não sendo possível assim estabelecer comunicação com o mesmo.

  Request body:

  {

```
    "nodeInformation": {
       "nodeId": [{
            "vNodeId": 12345678912345678123
       },
       {
            "vNodeId": 987654321987654321123
       }]
    }
}
```

- **Replace node**

  Em alguns casos um dispositivo pode apresentar algum tipo de anomalia, sendo nesses casos necessário proceder à troca do mesmo. Quando esta situação acontece é atribuído um novo *serial number* e *label key*, uma vez que o dispositivo é diferente do anterior. Deste modo é necessário recorrer à funcionalidade de *Replace* node para que as novas informações (*serial number*, *label key*) sejam atualizadas no sistema, até para a atribuição de um novo vNodeId.

  Request body:

```
{
    "nodeInformation": {
          "oldSerialNumber": 9898445566888,
          "vNodeId": 12345678912345678123
     }
}
```

- **Node information**

  Esta funcionalidade permite consultar as informações de um ou vários dispositivos, informações essas que foram definidas durante o processo de registo. Para obter estas informações, foram implementados dois mecanismos diferentes (*query method*). O primeiro é obter essas informações, enviando para o sistema o vNodeId ou vNodeId’s dos dispositivos dos quais queremos

obter informação. O segundo mecanismo é obter essas informações, mas desta vez fazendo a filtragem por userId. Isto só é possível caso já tenha sido gerado um *token* tendo em conta um userId e um vNodeId (este processo irá ser explicado posteriormente no capitulo 5.1.5 – Gestão de tokens).

Request body (usando vNodeId):


```
{
 "queryMethod": {
      "nodeId": [{
            "vNodeId": "12345678912345678123"
           },
           {
            "vNodeId": "98765432198765432112 3"
          }]
      },
     "entityId": "entity id"
}
```


- **Edit node**

  Esta funcionalidade, como o próprio nome indica, tinha como objetivo permitir que fosse possível editar algumas informações que foram atribuídas durante o processo de registo, tais como a *timezone*, latitude, longitude, entre outras.

  É importante referir que informações como o *serial number*, *label key* ou vNodeId não são possíveis de editar, uma vez que se tratam de informações únicas de cada dispositivo.

Request body:


```
{
  "nodeInformation": {
      "vNodeId": 1234567812345678123,
```

```
        "newTimeZone": "(GMT) Western Europe
        Time,London,Lisbon,Casablanca,Greenwich",
        "newGeoLat": 38.7,
        "newGeoLong": -9.14,
        "newTag": "Tags",
        "newName": "Name"
    }
}
```

- **Request System File**

  Os dispositivos IoT que a Domatica dispõe são compostos por inputs e outputs onde é possível ligar um conjunto variado de conectores e devices. Estes *devices* podem ser por exemplo, leds, sensores de temperatura, CO_2, humidade ou certos *relays* que permitem controlar um conjunto variado de outros *devices*.

  No entanto e segundo como foi elaborado o sistema presente nestes dispositivos, é necessário configurar os mesmos recorrendo à Domatica Tool. Esta ferramenta, desenvolvida pela empresa, tem como objetivo implementar as configurações básicas dos dispositivos. Nesta ferramenta é possível por exemplo visualizar todos os controladores e *devices* interligados a certo dispositivo, onde podemos configurar o nome a que queremos dar a cada controlador ou *device*. Imaginando por exemplo, que no dispositivo IoT temos interligado um sensor de temperatura, podemos definir o nome deste *device* como – "TEMP_SENSOR_01", sendo este nome conhecido dentro do sistema da empresa como 'idname'.

  Após a configuração de todos os controladores, *devices* por parte dos clientes ou utilizadores, é gerado um ficheiro XML – System File -, onde fica guardadas todas estas configurações e onde é possível estabelecer-se uma referência entre os *devices* físicos conectadas com o dispositivo.

  Deste modo, o principal objetivo desta funcionalidade era de poder comunicar com um dispositivo e aceder ao *System File* do mesmo.

  Como o *System File* é um ficheiro XML, era impraticável acedê-lo no seu formato original e devolver de volta ao utilizador, uma vez que as velocidades de comunicação iam ser afetadas por se tratar de um ficheiro

pesado para este tipo de comunicação. Assim foi necessário arranjar uma alternativa. Essa alternativa passou por converter o *System File* para um formato comprimido – gzip -, antes do mesmo ser enviado de volta para o utilizador.

O gzip implementa um algoritmo de compressão, uma variação do algoritmo já existe, o LZ77. Durante o processo de compressão, este algoritmo faz uma busca por *strings* duplicadas. Quando encontra uma *string* duplicada, a mesma é substituída por um ponteiro que têm como referência à primeira *string*, tornando assim o ficheiro mais pequeno e leve [41]. Com este algoritmo de compressão é possível então a transferência de ficheiro via web, salvaguardando a velocidade de comunicação.

Request body:


```
{
      "vNodeId": 1234567812345678123,
      "parameters": {}
}
```


- **Request User Program File**

O *User Program File* é também um ficheiro XML. Este ficheiro XML, à semelhança do *System Fil*e é também configurado através da Domatica Tool. O principal objetivo deste ficheiro é configurar um conjunto de regras para o dispositivo IoT. Estas regras podem ter um conjunto variado de funcionalidades. É através do *User Program File* que o utilizador pode configurar por exemplo que um alarme seja disparado quando a temperatura exceda um valor previamente definido pelo mesmo.

O objetivo desta funcionalidade passava por comunicar com o dispositivo e assim obter o *User Program File* que se encontrava no mesmo. Por se tratar de um ficheiro em formato XML, o mesmo era também comprimido com recurso ao gzip.

Request body:


```
{
   "vNodeId": 1234567812345678123,
```

```
  "maxWaitTime":300,
  "parameters": {}
}
```

- **Deploy User Program**

  Foi desenvolvida também a funcionalidade que permita fazer o *deploy* (envio) de um *User Program* para o dispositivo. Este ficheiro tinha de ser previamente comprimido em formato gzip antes do seu envio. Quando o mesmo chegasse à CORE API, era compilado e caso a compilação ocorresse sem qualquer erro o mesmo era enviado para o dispositivo. Tanto a compilação como a própria operação de envio para o dispositivo eram efetuadas consoantes as funções disponibilizadas pelo SDK.

  Request body:

```
{
  "vNodeId": 1234567812345678123,
  "datatype": "base64XMLProgram",
  "data": "YmFzZTY0IHVzZXIgcHJvZ3JhbSB4bWw=",
  "maxWaitTime":300,
  "parameters": {}
}
```

- **Get**

  Uma das mais-valias desta CORE API era a possibilidade de recorrendo a uma conexão de internet, comunicar com um dispositivo IoT independentemente da sua localização, e assim se proceder à leitura, monitorização ou controlo de um conjunto variado de *devices*.

  Esta funcionalidade permitia que o utilizador pudesse de forma simultânea obter o estado, naquele preciso momento, de um conjunto variado de *devices*, deste o valor da temperatura ao estado atual de lâmpada (ligada ou desligada). Como já foi referido anteriormente o dispositivo IoT estava configurado de modo a que houvesse uma relação com os *devices* físicos instalados no mesmo. Cada um dos *devices* era configurado com um 'idname'. Assim para

comunicar com um *device* e obter o valor ou estado atual do mesmo, era necessário que no pedido feito à camada L1 (e posteriormente à camada L0), fosse preciso incluir o 'idname' do *device*(s) do qual pretendia obter informação. A consulta deste(s) 'idname' podia ser feita através do *System File*. É possível também usa um método de autenticação diferente para cada pedido distinto à *gateway*. Os valores assinalados a amarelo representam o *id* virtual de cada dispositivo para o qual se pretende comunicar (vNodeId).

Request body:


```
"nodes": {
   "121312312312": {
       "authenticationMethod": {
           "gToken": {
              "token":"AABBCCDDEEFF990011233445566",
               "vNodeId": 12345678912345678123
           }
       },
       "objects": {
          "DCS_TEMPERATURE_LEIRIA.instShowValue":""
       }
   },
   "7682642342432423": {
       "authenticationMethod": {
           "userAuthentication": {
               "userEmail": "user@mail.com",
               "userPassword": "userPassword"
           }
       },
       "objects": {
          "DCS_TEMPERATURE_LISBON.instShowValue":""

       }
   }
 }
```

}

- **Set**

Esta funcionalidade é muito semelhante à funcionalidade do *Get*, sendo que neste caso o objetivo não era a leitura de um valor ou estado, mas sim de alterar um valor ou estado de um determinado *device*. O mesmo era feito novamente com recurso ao 'idname'. De modo a facilitar o uso desta API por parte do utilizador, tanto o formato em JSON para proceder ao *Get* ou ao *Set*, eram exatamente iguais, sendo necessário apenas inserir um valor na operação Set. É igualmente possível também o uso de método de autenticação diferente para cada pedido distinto à *gateway*. Os valores assinalados a amarelo representam o *id* virtual de cada dispositivo para o qual se pretende comunicar (vNodeId).

Request body:

```
{
"nodes": {
   "121312312312": {
       "authenticationMethod": {
           "gToken": {
       "token": "AABBCCDDEEFF99001122334455667788",
               "vNodeId": 12345678912345678123
           }
       },
       "objects": {
           "DCS_LEIRIA.CONTROL_ON_OFF": "1"
       }
   },
    "7682642342432423": {
       "authenticationMethod": {
           "userAuthentication": {
               "userEmail": "user@mail.com",
               "userPassword": "userPassword"
```

```
                }
            },
            "objects": {
                "DCS_LISBON.CONTROL_ON_OFF": "0"
            }
        }
    }
}
```

- **Request Logs**

  Um dos grandes desafios do mundo IoT, além do controlo de um conjunto variado de sensores, controladores, etc., era a possibilidade de o utilizador poder monitorizar todos os eventos desencadeados. Atualmente no mundo da indústria, muitas são já as empresas interligadas com dispositivos IoT, usando para monitorizar todos os eventos gerados pelos mesmos, desde mudanças de temperatura, de sensores fotoelétricos, controladores, etc.

  Deste modo, os dispositivos IoT da Domatica tinha a capacidade de armazenar milhares de *logs* (aproximadamente 155.000). Estes *logs* eram gerados sempre que era desencadeado um evento (acender ou apagar uma luz). Por outro lado, era também possível gerar *logs* de x em x tempo (por exemplo 5 em 5 minutos) de todos ou só de alguns *devices* que o utilizador achasse mais relevante para a sua solução.

  O objetivo desta funcionalidade passava então por permitir que utilizador pudesse consultar os *logs* gerados pelo dispositivo. Como já foi referido os dispositivos IoT da Domatica apenas tinham capacidade de armazenar aproximadamente 153.000 registos, sendo os mesmos guardados em formato csv. Quando este valor era atingido, e sempre que era gerado um novo log, o mesmo era escrito por cima do log mais antigo, e assim sucessivamente, procedendo-se então a este ciclo. Um dos grandes objetivos deste projeto era de auxiliar os dispositivos IoT de forma a que o utilizador pudesse consultar os *logs* sem que se visse limitado pela capacidade do dispositivo. Assim todos os *logs* gerados eram guardados na base de dados MySQL. O *request* dos *logs* podia ser efetuado consoante algumas opções. O utilizador podia obter

os últimos x *logs* (ou seja, os x mais recentes), entre dois períodos de tempo ou deste determinado tempo até ao tempo atual de quando efetuo o pedido.

Imaginando que era efetuado um primeiro *request* de *logs* (os últimos 20) para um determinado dispositivo, era necessário comunicar com o mesmo e assim obter todos os *logs* disponíveis pelo mesmo, guardando-os na base de dados. Após isto, era feita uma *query* à base de dados de modo a ir obter apenas os últimos 20 e envia-los ao utilizador. Caso fosse efetuado um novo *request* ao dispositivo, pedindo por exemplo os últimos 100 logs era feito inicialmente uma consulta à base de dados de modo a confirmar o tempo atual (*timestamp*) em que era efetuado o pedido era igual ao último *timestamp* dos logs guardado na base de dados. Se fosse, então não era necessário comunicar com o dispositivo, uma vez que a base de dados tinha todos os dados necessários para o utilizador. Caso contrário era necessário comunicar com o dispositivo, mas neste caso não era necessário obter todos os *logs*, apenas os que faltavam para completar a informação pedida pelo utilizador, ou seja a partir do último *timestamp* disponível até ao tempo atual. Se os *logs* estivessem configurados para serem gerados de 5 em 5 minutos e que o utilizador pedia os últimos 20 *logs*, o mesmo podia definir uma tolerância. A tolerância servia para casos em que pedindo os últimos 20 *logs* e faltando os últimos 5 *logs* (gerados nós últimos 5 minutos), se fosse definida uma tolerância de 10 minutos, não era necessário comunicar com o dispositivo, porque mesmo faltando os últimos 5 minutos na base de dados, foi definida uma tolerância de 10 minutos. É importante referir que a operação de comunicar com o dispositivo e obter os seus *logs* pode ser uma operação com grandes custos em questões de tempo de processamento. Isto acontece porque é necessário comunicar com o dispositivo e que esta recolha todos os *logs*, e dependendo do seu número, este processo poderá demorar algum tempo. Por outro lado, após obter os *logs* do dispositivo, era necessário guardar os mesmos na base de dados, podendo também haver algum *delay*, uma vez que trata da inserção de milhares de linhas ao mesmo tempo. Assim o utilizador poderia definir no seu pedido uma variável – maxWaitTime (por exemplo 300 segundos/5 minutos). Este valor informava a CORE API de quanto tempo o utilizador pretendia ficar à espera da resposta do seu pedido. Caso o

pedido já estivesse a ser processado há mais tempo que o definido no maxWaitTime, o mesmo era informado ao utilizador. Neste caso o utilizador teria de efetuar posteriormente um novo pedido para verificar se o seu pedido já tinha sido processado. Para esta operação é necessário executa-la numa *thread* externa de modo, que enquanto estão a ser processados os pedidos de *logs*, a CORE API seja capaz, sem qualquer tipo de limitação, de continuar a processador outros pedidos do mesmo utilizador ou de outros, sejam esses pedidos de *logs* ou de outra natureza.

Request body (os últimos 150000 logs):

```
{
      "vNodeId": 1234567812345678123,
      "maxWaitTime": 300,
      "parameters": {
            "quantity": "150000",
            "tolerance": "300"
      }
}
```

Request body* (entre duas datas*):

```
{
     "vNodeId": 123456781234567812312З,
     "maxWaitTime": 300,
     "parameters": {
             "beginTimestamp": "146376303",
             "endTimeStamp": "169377375",
             "tolerance": "300"
      }
}
```

- **Deploy System Program**

}

Os dipositivos IoT da empresa tem um conjunto de ficheiros de configuração bem como alguns drivers – A que se dá o nome de *System Program* -, que são imprescindíveis para o seu funcionamento. Esta funcionalidade tinha como objetivo o de permitir a configuração desse mesmo *System Program*. Estas configurações estão usualmente guardadas dentro de uma pasta. Para esta funcionalidade, essa mesma pasta, em formato zip, tinha de ser comprimida para um formato em que fosse possível enviar REST (à semelhança do que já se sucedeu com alguns ficheiros). Desta feita recorreu-se ao algoritmo de compressão – base64. A camada L1 antes de estabelecer a comunicação com a Core API, era responsável por criar uma pasta temporária, proceder à descompressão da pasta do *System Program* (de base64 para .zip) e colocar a mesma, dentro da pasta temporária anteriormente criada. A camada L1 era também responsável, após este processo, de estabelecer a habitual comunicação com a camada L0 e de lhe enviar o caminho de onde se encontrava a pasta temporária no sistema. Após a camada L0 receber o caminho da pasta, era responsável por proceder ao *unzip* do *System Program*. Dentro da raiz desta pasta, era possível encontrar um ficheiro (project.def) configurado pelo utilizador. Neste ficheiro estavam configuradas algumas variáveis, como por exemplo o caminho dentro da pasta do *System Program*, para as outras pastas necessárias para efetuar a operação de *deploy*, uma vez que o utilizador poderia colocar o conjunto de drivers dentro de qualquer pasta ou sem qualquer nome em específico.

Request body:


```
{
  "vNodeId": 1234567812345678123,
  "datatype": "base64XMLProgram",
  "data": "YmFzZTY0IGlwZng=",
  "maxWaitTime":300,
  "parameters": {}
}
```


5.1.5 Gestão de Tokens

Os *tokens*, tal como já foi referido, era usado como um dos métodos de autenticação disponíveis para os utilizadores. Assim foram desenvolvidas algumas funcionalidades que servissem de suporte para a gestão desses mesmos *tokens*. Usualmente os *tokens* eram usados apenas para operações que envolvessem os dispositivos IoT, uma vez que os mesmos eram gerados tendo em conta o userId e o vNodeId.

- **Add Token**

  Esta funcionalidade permitia que fossem adicionados novos *tokens*. Para a criação desses *tokens* era necessário o userId do utilizador bem como o vNodeId do dispositivo. Os *tokens* eram gerados com através de num algoritmo.

  Request body:


```
{
  "tokenInformation": {
       "userId": "user1",
       "vNodeId": 1234567812345678123,
       "name": "token name"
    }
}
```


- **Remove Token**

  Esta funcionalidade permitia que após a criação de um *token*, o mesmo pudesse ser removido do sistema e assim não sendo possível mais a sua utilização.

  Request body:


```
{
   "tokenInformation": {
       "token": "HDASI74732GHDJHDA63FFG2",
       "vNodeId": 1234567812345678123
    }
}
```

- **List Tokens**

  Esta funcionalidade permitia que um utilizador pudesse consultar todos os seus *tokens*, filtrando-os por vNodeId. Deste modo, para proceder a este pedido era necessário enviar o conjunto de vNodeId's bem como o userId, associado às varias vNodeId's.

Request body:


```
{
 "nodesInformation": {
      "nodeId": [{
           "vNodeId": 12345678912345678123
      },
      {
           "vNodeId": 98765432198765432123
      }]
 },
 "userInformation": {
      "userId": "user Id"
 }
}
```


5.1.6 Armazenamento dos dados

Como foi possível observar durante a descrição das funcionalidades implementadas, a Core API está constantemente a processar vários tipos de dados. Muitos destes dados foram armazenados numa base de dados (MySQL) de modo a que os mesmos pudessem ser acedidos sempre que necessário. Os *logs* gerados pelos vários dipositivos eram um dos fluxos de dados com maior ocorrência na base de dados. Isto acontecia uma vez que, por exemplo, os dispositivos proprietários da Domatica tinham uma quantidade limitada de armazenamento interno para os *logs* (cerca de 155 mil). Deste modo, era necessário que os mesmos fossem armazenados numa estrutura externa ao dispositivo para assim o seu histórico estar sempre salvaguardado e acessível desde sempre, não havendo assim perdas de histórico, quando o dispositivo atingisse o seu limite de armazenamento.

Por outro lado, é expectável que o número de pedidos à Core API tenham tendência para aumentar, o que irá significar num maior número de fluxo de dados, não podendo de forma alguma a velocidade de processamento ser afetada. Existem certos tipos de dados/informações que estão constantemente presentes em grande maioria dos pedidos efetuados, como é caso do vNodeId ou do *serial number*, ou até mesmo dos *tokens* e dos *sessionId's* usados nos mecanismos de autenticação. Nestes casos, estas informações estavam armazenadas também na memória cache da Core API. Assim sempre que era necessário utilizar estes dados para certas operações, os mesmos podiam ser acedidos na memória cache e assim aumentar a velocidade de processamento do pedido, uma vez que nestes casos não era necessário estabelecer conexão à base de dados.

A base de dados foi criada e configurada com recurso ao *software* MySQL Workbench. O Workbench é uma ferramenta visual, fornecendo ferramentas de modelação dos dados, desenvolvimento SQL e de administração, para configurações no servidor, administração de utilizadores, entre outras. Para o caso concreto deste projeto foram criadas seis tabelas relacionais MySQL: *Users, Nodes, Entity, Tokens, Logs e Hardware Keys*. Esta última tabela era responsável por guardar os *registrations codes* gerados necessários às operações de *Claim* e *Release*.

5.2 Testes e Ferramentas

Neste capítulo irão ser descritos os vários testes efetuados à solução final, bem como algumas ferramentas que foram usadas como auxílio a esses mesmos testes.

Para o correto funcionamento da solução e deste modo a mesma entrar produção, é necessário testa-la de forma a garantir a sua integridade. É necessário garantir que a solução é confiável, ou seja, que não apresenta qualquer tipo de falha durante a sua execução. Por outro lado, em termos de funcionalidade, para que a solução tenha o comportamento esperado e definido nos requisitos, e também em termos de performance, para que a solução tenha um tempo de resposta aceitável, mesmo quando se encontra com um grande volume de processamentos. Assim e para garantir os três principais pontos descrito anteriormente, foram desenvolvidos alguns testes à solução e que serão descritos de seguida.

5.2.1 Testes Unitários

Camada L0

Na camada L0 eram processados todos os tipos de pedidos. Como já foi referido no capítulo 4.2.3 Camada L0, esta camada foi desenvolvida em C++, havendo assim uma grande gestão da memória alocada. Em certos casos, poderão existir algumas falhas no que toca à libertação dessa memória alocada, o que iria originar *memory leaks*. De forma a garantir a não existência de fugas, foi utilizada a ferramenta de testes Valgrind. Esta ferramenta era executada ao mesmo tempo que a aplicação e tinham como objetivo detetar erros decorrentes do uso incorreto da memória. Para isso, utilizava um conjunto de ferramentas que codificavam o código da solução para uma representação específica do Valgrind. Este facto levava a uma perda de desempenho da solução, no entanto esta perda de desempenho não foi tida em conta, primeiro porque era normal no funcionamento desta ferramenta e por outro porque o principal objetivo era detetar possíveis fugas de memória.

Após a execução do programa, o Valgrind gerava um ficheiro de .txt onde descrevia para funcionalidade testada, quais e quantos os erros de gestão de memória encontrados, bem como o número de bytes alocados e o número de bytes que ficaram por libertar.

Com o uso desta ferramenta foi possível garantir a integridade e o funcionamento da camada L0, de modo a que esta não apresentasse futuros problemas de funcionamento, quando a mesma já se encontrasse a ser usada pelos diversos utilizadores.

Camada L1

Os testes unitários são bastante importantes de modo a validar cada uma das funcionalidades desenvolvidas. Este tipo de teste, tem como objetivo garantir que

determinada funcionalidade tenha o comportamento expectável e também para validar que quando uma nova funcionalidade é desenvolvida, a mesma não venha a afetar o funcionamento das funcionalidades já desenvolvidas e implementadas. Estes testes são habitualmente executados em ambientes controláveis, onde todos os dados utilizados são exclusivamente usados para cada teste, sendo para isso criados diferentes tipos de dados.

No caso mais concreto do projeto, um dos principais objetivos era garantir que os dados em JSON, que eram transferidos da camada L1 para a camada L0, tinha a formatação correta e previamente definida para cada uma das funcionalidades. Para isso foi criada, na camada L1, uma classe de testes, onde foram construídas várias mensagens em JSON de forma a simular dados para cada uma das funcionalidades. Estas mensagens tinham como principal objetivo testar que cada funcionalidade na camada L1 (por exemplo a operação de New User) tinha sido desenvolvida de acordo com a correta sintaxe da mensagem JSON. Assim cada mensagem JSON foi escrita exatamente com a mesma sintaxe previamente definida para cada funcionalidade. Após esta fase, cada uma das mensagens era utlizada para a funcionalidade correspondente, para assim garantir que a mesma foi desenvolvida da forma correta.

5.2.2 Testes de aceitação

Após garantir o correto funcionamento de cada funcionalidade em termos de código, era necessário garantir também que a solução já se encontrava pronta a ser usada pelos utilizadores finais. Para isso foram desenvolvidos alguns testes de aceitação.

Antes do desenvolvimento das funcionalidades implementadas, alguns colaboradores da empresa, já tinham feedback de alguns clientes relativamente ao tipo de funcionalidades a desenvolver e também qual o comportamento esperado para cada uma delas. Para isso os testes de aceitação foram desenvolvidos junto desses mesmos colaboradores da empresa. Com a realização destes testes, foi possível recolher algumas informações:

- Falhas encontradas;
- Dificuldade de utilização;
- Tempo de resposta;
- Possíveis melhorias;

Resultados

Com as informações recolhidas, foi possível obter alguns resultados e assim aferir sobre o funcionamento da solução.

Os colaboradores/utilizadores que realizaram estes testes, consideraram que a solução era simples de utilizar, intuitiva, não encontrando qualquer tipo de falhas, ficando globalmente satisfeitos com o seu funcionamento.

Alguns dos colaboradores/utilizadores, consideraram que em algumas funcionalidades, como é o caso do *Request Logs* ou do *Deploy System Program*, a solução deveria de informar o utilizador do tempo de resposta expectável ou informar o utilizador que o pedido esta a demorar mais do que o expectável. Isto porque, este tipo de operações, podem ser operações onde existe uma grande quantidade de processamento de dados, podendo demorar algum tempo. Por exemplo, quando não existe na base de dados ainda qualquer histórico do dispositivo, é necessário comunicar com o mesmo de forma a ir buscar todos os *logs* disponíveis. Caso o dispositivo, já tenha num número considerável de *logs* (cerca de 150 mil), o pedido irá demorar mais que o expectável, havendo casos em que o dispositivo demora cerca de 40 segundos a devolver todos os *logs*. Nestes casos, o utilizador não tem qualquer feedback sobre a operação, não sabendo o estado da mesma.

5.3 Segurança

A segurança hoje em dia no mundo tecnológico é um dos aspetos mais importantes a ter consideração no desenvolvimento de qualquer tipo de solução. As falhas de segurança ou a inexistência de qualquer medida de segurança acarreta consigo problemas graves, fazendo que os utilizadores finais não sintam a devida confiança em utilizar a solução desenvolvida, tornando-a mais vulnerável a possíveis ataques informáticos.

Por outro lado, a segurança no mundo IoT, é uma das mais recentes preocupações tecnológicas ao nível de segurança. É necessário que os dados de uma infraestrutura IoT sejam protegidos na *cloud*, durante a comunicação e ao nível dos dispositivos inteligentes.

O ano de 2017 registou até ao momento o maior número de ataques a dispositivos IoT, segundo uma investigação levada a cabo por investigadores da Kaspersky Lab [42]. Devido a este crescente número de ataques, a IoT tornou-se um dos maiores atrativos para os ciber criminosos, uma vez que ao conseguirem piratear este tipo de dispositivos, conseguem ter controlo sob os mesmos e sob os utilizadores dos mesmos. Se pensarmos num edifício controlado por dispositivos IoT, desde portas, janelas, sistemas de

refrigeração, de luminosidade, entre outros, ao ser atacado, o mesmo fico comprometido e o seu normal funcionamento fico afetado. Um ataque a este nível iria comprometer a segurança tanto do edifício, mas também dos seus utilizadores, uma vez que os mesmos poderiam ficar retidos dentro do edifício (caso as portas fossem por exemplo trancadas à distância pelo hacker) ou impossibilitados de usufruir das instalações. Também já foram noticiadas situações em que um conjunto de dispositivos IoT de diversos utilizadores, desde torradeiras, frigoríficos, máquinas de lavar, entre outros, foram usados em simultâneo para efetuar ataques informáticos, sem que os utilizadores dos mesmos tivessem essa perceção. Isto acontece porque qualquer dispositivo ligado à internet torna-se mais vulnerável a qualquer tipo de ataque, e sendo os dispositivos IoT, dispositivos constantemente ligados à internet, tornam-se assim mais apetecíveis para este tipo de ataques. Deste modo, o crescente aumento destes dispositivos no mercado, acarreta com ele, uma constante preocupação ao nível de segurança.

Neste capítulo serão descritas algumas medidas de segurança implementadas no projeto de modo a torna a solução mais segura e robusta.

5.3.1 Ligação HTTPS

Qualquer ligação/comunicação entre dois sistemas é sem dúvida alguma um ponto sensível, onde o foco principal passa pela privacidade e integridade dos dados durante a comunicação.

Para isso, uma das principais medidas passa pela encriptação dos dados durante a sua transmissão. Assim, a utilização do protocolo HTTPS (Hyper Text Transfer Protocol Secure) ao invés do HTTP (Hyper Text Transfer Protocol), era a opção mais válida a usar durante a transmissão dos dados de modo a que a encriptação dos mesmos fosse garantida, como é possível observar na Figura 22 - HTTP vs HTTPS.

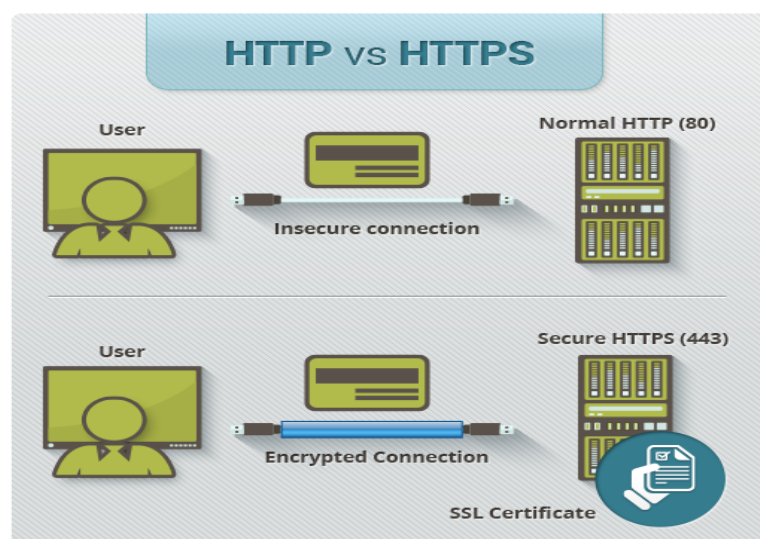

Figura 22 - HTTP vs HTTPS

O protocolo HTTPS é uma versão segura do protocolo HTTP. Este protocolo utiliza dois outros protocolos para garantir a criptografia dos dados – SSL (Secure Sockets Layer) e o TLS (Transport Layer Security). Tanto o SSL como o TLS usam uma infraestrutura de chaves públicas assimétrico. Neste tipo de sistemas são usadas duas chaves, uma pública e uma privada. Qualquer dado que seja encriptado, o mesmo só pode ser descodificado pela chave privada e vice-versa. A chave privada é mantida protegida e a mesma só pode ser acessível pelo proprietário da mesma. Por outro lado, a chave pública é distribuída a todos que necessitam da mesma e que tenham capacidade de decifrar os dados cifrados com a chave privada [43].

Este tipo de segurança com recurso ao HTTPS tem como principal objetivo prevenir ataques do tipo *man-in-the-middle* – Figura 23 - Man-in-the-middle.

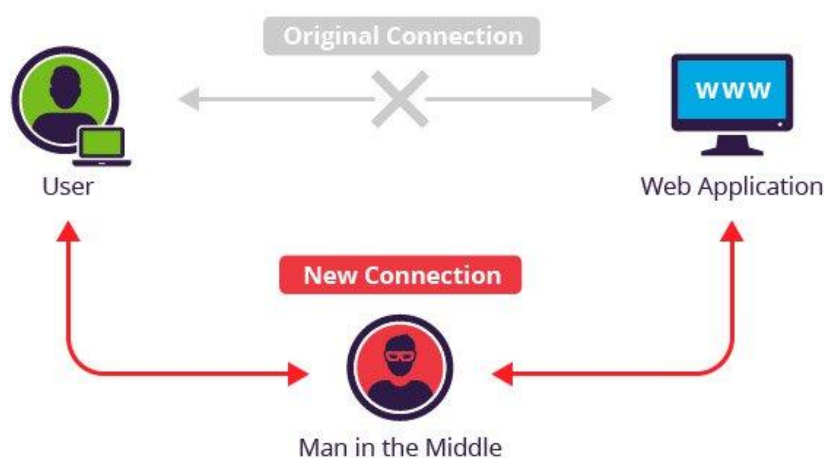

Figura 23 - Man-in-the-middle

Estes tipos de ataques caracterizam-se essencialmente quando alguém mal-intencionado consegue intercetar os dados transmitidos, podendo os mesmos ser alterados sem que o utilizador se aperceba, antes que os dados cheguem ao destino final.

Assim o protocolo HTTPS foi utilizado de modo a garantir a integridade e segurança dos dados transmitidos entre camada L1 e a camada L0 (Core API). Por outro lado, em alguns dos *requests* era necessário que a Core API promovesse a comunicação com os

dispositivos IoT. Neste caso, a Domatica implementou um *tunnel* seguro com um protocolo de comunicação específico para que fosse possível a comunicação com os dispositivos IoT sem que esta comunicação fosse de alguma forma também comprometida, permitindo assim a integração e segurança dos dados.

5.3.2 Core API

Como já foi explicado anteriormente (Capítulo 6.5.1 Ligação HTTPS), a comunicação entre a camada(s) L1 e a camada L0 (Core API), bem como com os dispositivos IoT, encontrava-se protegida. A Core API respondia a pedidos HTTP (como por exemplo o POST) e deste modo para aumentar o nível de segurança foi definido que não seria implementado nenhum método HTTP GET. Esta decisão prendeu-se pelo o facto que em métodos GET os dados a enviar e/ou a receber são incluídos no URL do pedido a efetuar, e deste modo poderiam ser mais facilmente acedidos por qualquer utilizador, podendo este ficar com acesso a esses mesmos dados quando pertencentes a outro utilizador. Assim todos os *request* foram implementados com o método POST, onde dos dados eram enviados no corpo do *request*, não ficando assim visíveis no URL – Figura 24 - GET vs POST.

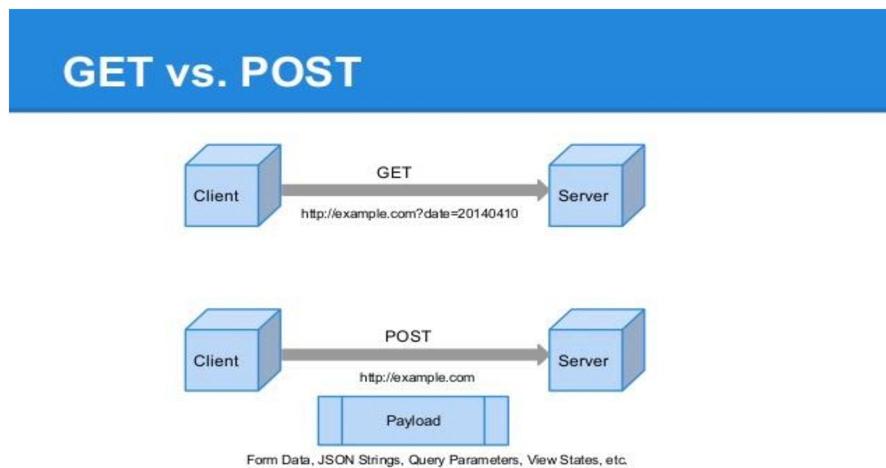

Figura 24 - GET vs POST

Autenticação

Para garantir uma maior segurança, foi implementado um sistema de autenticação, onde foram desenvolvidos 3 diferentes tipos de autenticação – Capítulo 6.3.1 Métodos de autenticação. A implementação de métodos de autenticação tem como principal objetivo validar a identidade digital do utilizador que tenha efetuado determinado pedido. Deste

modo para qualquer *request* era necessário usar obrigatoriamente um dos métodos de autenticação disponíveis, permitindo assim que os mesmos *requests* fossem efetuados por utilizadores registados e com autorização para tal. Dos 3 métodos de autenticação implementados, o método de autenticação por *token* era provavelmente o mais seguro a usar, ao invés por exemplo do habitual método de autenticação por email/password. Um *token* é uma chave única que identifica um determinado utilizador (e neste caso associado também a um dispositivo IoT), e que é caracterizada por ter um determinado tempo de validade. Após este tempo, o *token* é eliminado do sistema não podendo ser mais usado. Este método de autenticação, à semelhança do método de autenticação por sessionId, são métodos mais eficazes no que diz respeito à segurança, uma vez que um possível roubo dessas chaves de autenticação, a probabilidade de acesso e roubo aos dados era menor, uma vez que apresentavam um tempo de uso/validação limitado. O mesmo não acontece num método de autenticação mais convencional, como é o caso do método de autenticação por *email/password*. Nesta situação, caso haja roubo das credenciais de acesso do utilizador, as mesmas poderão ser usadas por tempo ilimitado até que seja detetada a fraude, algo que não acontece com o sessionId ou o *token*, devido ao seu tempo limitado de uso.

Acesso aos dados

Outro mecanismo de segurança implementado foi o acesso aos dados por diferentes tipos de utilizadores. Em primeiro lugar, um utilizador não poderá por exemplo visualizar os dados de outro utilizador. Em segundo lugar, certos *requests* estavam apenas permitidos a utilizadores que tivessem níveis de privilégios para tal. Deste modo haviam certas operações que só alguns utilizadores poderiam efetuar (por exemplo um utilizador *Admin* ou *Owner*), sendo assim garantido mais um mecanismo de segurança.

5.3.3 Base de dados

Como já foi referido, foi implementada uma base de dados MySQL para que fossem armazenados os diferentes tipos de dados necessários ao funcionamento da Core API.

Por defeito, o MySQL não usa encriptação nas ligações entre o cliente e o servidor, uma vez que tornaria essas ligações muito mais lentas. Assim qualquer utilizador pode observar o tráfico e assim ter acesso aos dados recebidos e enviados. Alguns destes dados eram considerados sensíveis, como por exemplo as *passwords* dos utilizadores. O MySQL

desde a versão 4.0.0, teve em consideração esta situação, e implementou o suporte para o protocolo SSL.

Por outro lado, foi necessário que alguma da informação na base de dados fosse guardada de forma encriptada, de modo a garantir a integridade e segurança dos dados, não permitindo o seu acesso e uso por parte de utilizadores mal-intencionados.

Foram também tomadas outras opções para reforçar a segurança. A base de dados foi instalada numa máquina diferente da máquina onde esta a correr a Core API e foi também alterada a porta padrão que usualmente o MySQL usa para as suas conexões.

6. Conclusão

Atualmente a quantidade de dispositivos IoT inteligentes tem tido um crescimento exponencial, quer no nosso dia-a-dia quer no mercado tecnológico, sendo cada vez mais a quantidade de pessoas, empresas e fabricantes que se tem debruçado e investido nesta área. No entanto, ainda se sentem algumas dificuldades no que toca ao desenvolvimento e integração de soluções IoT, devido principalmente ao elevado tempo necessário de desenvolvimento, complexidade e falta de conhecimento. O desenvolvimento deste projeto passava primordialmente por facilitar esse mesmo desenvolvimento para que de uma forma simples e intuitiva fosse possível comunicar, monitorizar e controlar um variado número de dispositivos IoT.

Como referido inicialmente neste documento, o objetivo deste projeto passou pelo desenvolvimento de uma IoT Core API. Esta Core API foi integrada na Domatica Quantum *cloud*, ficando esta dotada de um conjunto de funcionalidades e serviços que promoviam essencialmente a comunicação via WEB com dispositivos IoT inteligentes.

Estes dispositivos inteligentes podiam ser dispositivos proprietários da Domatica Global Solutions ou outro tipo de dispositivos não proprietários, como é o caso do Raspberry PI, routers, *switches* ou outro qualquer tipo dispositivo inteligente conectado à internet. Por outro lado, foi necessário desenvolver também uma RESTful API que promovesse a comunicação entre os utilizadores e a Core API, usando o JSON como protocolo de mensagens, com recurso ao *brocker* RabbitMQ para promover essa transferência de mensagens entre camadas.

Durante a fase de desenvolvimento, foram sentidas algumas dificuldades, o que é normal tendo em conta a falta de experiência no desenvolvimento de projetos desta envergadura e de conhecimentos no que toca à área da *Internet Of Things.* No entanto, em regra geral, essas dificuldades foram ultrapassadas, tendo sido adquiridos conhecimentos sem dúvida algumas imprescindíveis, que contribuíram tanto para o meu crescimento pessoal, mas essencialmente profissional.

Assim foi possível desenvolver uma solução que se caracteriza por ser simples e intuitiva de usar, quebrando a barreira existente na complexidade de comunicação com dispositivos IoT. Com esta solução o utilizador tem a possibilidade de comunicar com o seu dispositivo IoT, podendo assim monitorizar os valores lidos pelos diversos sensores a ele ligados (temperatura, humidade, CO_2, etc.) e ter controlo sobre diversos objetos (por

exemplo uma lâmpada, um dispositivo de rega, parar o funcionamento de uma máquina industrial, etc). Por outro lado, poderá também monitorizar todos esses sensores e objetos, tendo acesso a todos as mudanças de valores e estados. Com isto é possível por exemplo, integrar esta solução numa fábrica industrial, em tempo real a controlar o funcionamento das suas linhas de produção, temperatura das máquinas, forçar paragens, controlar o número de peças fabricas e também aferir por exemplo os tempos de paragens de determinada linha de produção. Todos estes cenários e muitos outros, poderão ser acedidos em tempo real em qualquer lugar desde que haja conexão à internet. Também com esta solução é possível configurar à distância qualquer um dos dispositivos IoT, que esteja assim, conectados à Domatica Quantum Cloud. A solução foi também alvo de alguns tipos de testes, de forma a melhorar o seu funcionamento e garantir que o resultado final foi o delineado nos requisitos.

Em suma, o mundo IoT esta cada vez mais presente no nosso quotidiano, seja por lazer ou por trabalho. Com este tipo de soluções, o mercado industrial tem sido sem dúvida alguma um dos mercados que mais tem investido nesta área, permitindo assim dotar as suas tecnologias com uma camada inteligente, melhorando a sua eficiência e gastos. No entanto é também importante referir que a segurança neste tipo de soluções e dispositivos, tem um fator primordial e de bastante consideração, devido essencialmente ao seu rápido crescimento e por se inserirem cada vez mais nas nossas tarefas quotidianas.

6.1 Trabalho Futuro

Como em grande parte das soluções desenvolvidas, há sempre margem para melhoramentos ou para a implementação de novas funcionalidades. Tratando-se de uma solução IoT essa margem é ainda maior uma vez que é uma tecnologia emergente em constante crescimento.

Deste modo existem algumas funcionalidades a serem melhoradas futuramente. Dessas funcionalidades destacam-se por exemplo o Deploy System Program, onde futuramente poderá ser possível fazer o *deploy* desta configuração, não apenas para um dispositivo de cada vez, mas para vários, facilitando assim a experiência do utilizador, bem como o tempo despendido. Por outro lado, existem certas funcionalidades (por exemplo Request Logs), que devido ao seu elevado processamento, podem demorar mais que o expectável (sendo o expectável na ordem dos milissegundos). Nestes casos é necessário a implementação de *multithreading*, para que a Core API possa processar vários pedidos simultaneamente, não

deixando nenhum pedido em espera, enquanto outro pedido de elevada envergadura esta a ser processado.
Por outro lado, irão ser futuramente adicionadas novos serviços à Domatica Quantum Cloud, destacando-se por exemplo o FTP, MQTT, Dropbox, Twitter, Facebook, entre outros. O MQTT é neste momento um dos serviços com maior prioridade de desenvolvimento, uma vez que se trata de um dos protocolos IoT mais usados atualmente neste tipo de soluções.

Bibliografia

[1] E. Alecrim, 22 Fevereivo 2017. [Online]. Available: https://www.infowester.com/iot.php.

[2] J. Morgan, "forbes," 1 Março 2017. [Online]. Available: https://www.forbes.com/sites/jacobmorgan/2014/05/13/simple-explanation-internet-things-that-anyone-can-understand/#78f5eade1d09.

[3] "netscandigital," [Online]. Available: http://netscandigital.com/blog/internet-das-coisas/. [Acedido em 26 Fevereiro 2017].

[4] "domaticasolutions," [Online]. Available: http://domaticasolutions.com/. [Acedido em 8 Março 2017].

[5] "idgnow," [Online]. Available: http://idgnow.com.br/mobilidade/2016/06/01/dispositivos-de-iot-ultrapassarao-a-quantidade-de-celulares-em-uso-em-2018/ . [Acedido em 12 Março 2017].

[6] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris e David Orchard. [Online]. Available: https://www.w3.org/TR/ws-arch/. [Acedido em 22 Setembro 2017].

[7] T. Berners-Lee, Fielding R.T. e Larry Masinter, "Uniform Resource Identifier (URI): Generic Syntax," 2002.

[8] S. Vinoski, "REMOTE PROCEDURE CALLS," em *Internacional Software Development*, 2012.

[9] E. Christensen, F. Curbera, G. Meredith e S. Weerawarana. [Online]. Available: https://www.w3.org/TR/wsdl. [Acedido em 20 Setembro 2017].

[10] G. M. Waleed e R. B. Ahmad, "Security Protection using Simple Object Access Protocol (SOAP) Messages Techniques," em *International Conference on Electronic Design*, 2008.

[Online]. Available: https://www.ibm.com/developerworks/library/ws-soa-design1/. [Acedido em 19 Setembro 2017].
11]

R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” 2000.
12]

“tallan,” [Online]. Available: https://blog.tallan.com/2015/12/08/azure-iot-hub-vs-event-hub/. [Acedido em 17 Setembro 2017].
13]

“microsoft,” [Online]. Available: https://docs.microsoft.com/pt-pt/azure/iot-hub/iot-hub-what-is-iot-hub. [Acedido em 17 Setembro 2017].
14]

“robomq,” [Online]. Available: https://robomq.readthedocs.io/en/latest/. [Acedido em 18 Setembro 2017].
15]

“device hive,” [Online]. Available: https://devicehive.com/. [Acedido em 18 Setembro 2017].
16]

“developer.amazon,” [Online]. Available: https://developer.amazon.com/alexa. [Acedido em 17 Setembro 2017].
17]

“macchina,” [Online]. Available: https://macchina.io/. [Acedido em 12 Setembro 2017].
18]

“m2mlabs,” [Online]. Available: http://www.m2mlabs.com/. [Acedido em 17 Setembro 2017].
19]

“nodered,” [Online]. Available: https://nodered.org/. [Acedido em 16 Setembro 2017].
20]

“openconnectivity,” [Online]. Available: https://openconnectivity.org/. [Acedido em 17 Setembro 2017].
21]

“zettajs,” [Online]. Available: http://www.zettajs.org/. [Acedido em 18 Setembro 2017].
22]

“thingworx,” [Online]. Available: https://www.thingworx.com/. [Acedido em 18 Setembro 2017].
23]

R. A. Atmoko*, R Riantini e M K Hasin, “IoT real time data acquisition using

24] MQTT protocol," Indonesia.

[Online]. Available: http://sensetecnic.com/products-and-services/fred-build-iot-
25] apps-faster/. [Acedido em 25 Setembro 2016].

J. L. Fernandes, I. C. Lopes, J. J. P. C. Rodrigues e S. Ullah, "Performance
26] Evaluation of RESTful Web Services and AMQP Protocol".

"domaticasolutions," [Online]. Available:
27] http://domaticasolutions.com/quantum_node.php. [Acedido em 28 Março 2017].

"knx," [Online]. Available: https://www.knx.org/pt/knx/associacao/o-que-o-
28] knx/index.php. [Acedido em 29 Março 2017].

"simplymodbus," [Online]. Available:
29] http://www.simplymodbus.ca/faq.htm#Modbus. [Acedido em 27 Março 2017].

R. Zazueta, "mashery," 12 Agosto 2017. [Online]. Available:
30] https://www.mashery.com/blog/api-data-exchange-xml-vs-json.

M. Rouse, "whatis.techtarget.," [Online]. Available:
31] http://whatis.techtarget.com/definition/message-broker. [Acedido em 16 Agosto
2017].

J. Stansberry, "linkedin," [Online]. Available:
32] https://www.linkedin.com/pulse/iot-communication-protocols-james-stansberry.
[Acedido em 11 Julho 2017].

"devmedia," [Online]. Available: http://www.devmedia.com.br/introducao-ao-
33] amqp-com-rabbitmq/33036. [Acedido em 22 Junho 2017].

"digitalocean," [Online]. Available:
34] https://www.digitalocean.com/community/tutorials/an-advanced-message-queuing-
protocol-amqp-walkthrough. [Acedido em 29 Junho 2017].

"embarcados," [Online]. Available: https://www.embarcados.com.br/mqtt-
35] protocolos-para-iot/. [Acedido em 06 Julho 2017].

R. Jamie M e F. Jeremy G, "Sensor Networks and Grid Middleware for
36] Laboratory Monitoring," 2005.

[37] P. T. EUGSTER, P. A. FELBER, R. GUERRAOUI e A.-M. KERMARREC, "The Many Faces of Publish/Subscribe," 2003.

[38] X. Chen, "Constrained Application Protocol for Internet of," 2014.

[39] "educba," 2 Dezembro 2015. [Online]. Available: https://www.educba.com/high-level-languages-vs-low-level-languages/. [Acedido em 12 Agosto 2017].

[40] M. Rouse, "teachtarget," [Online]. Available: http://searchwindowsserver.techtarget.com/definition/memory-leak. [Acedido em 22 Agosto 2017].

[41] "gzip," [Online]. Available: http://www.gzip.org/. [Acedido em 28 Agosto 2017].

[42] L. Vedor, "pcguia," [Online]. Available: https://www.pcguia.pt/2017/07/ataques-malware-dispositivos-iot-duplicou-2017/. [Acedido em 13 Setembro 2017].

[43] "instantssl," [Online]. Available: https://www.instantssl.com/ssl-certificate-products/https.html. [Acedido em Setembro 11 2017].

[44] [Online]. Available: https://lists.oasis-open.org/archives/amqp/201202/msg00086/StormMQ_WhitePaper_-_A_Comparison_of_AMQP_and_MQTT.pdf. [Acedido em 12 Julho 2017].

Esta página foi intencionalmente deixada em branco

Domatica Quantum Cloud

L0 API

1. Authentication methods

1.1 User Authentication

To do the authentication by user authentication. This body **is used before (see the example on section 1.4)** the other data fields.

Body example:


```
{
    "authenticationMethod": {
            "userAuthentication": {
             "userEmail": "user@mail.com",
             "userPassword": "userPassword"
            }
       }
  }
```


1.2 GToken

To do the authentication by token. This body **is used before (see the example on section 1.4)** the other data fields.

Body example:


```
{
    "authenticationMethod": {
            "tokenAuthentication": {
            "gToken":"AABBCCDDEEFF990011223344556677
        88"
                "vNodeId": 12345678912345678123
```

```
            }
        }
    }
```

1.3 SessionId

1.3.1 Create sessionId

- This body example shows how to create one sessionId. In section 1.4 you can see how to use the sessionId.
- Notes:
- Use the Authentication Method - UserAuthentication -, to create the sessionId;
- Before using the sessionId authentication in requests you need to create one first;
- All sessionId have one expiration time (30 minutes). After this time, you need to create another one;
- Every time you use the sessionId within the validity time the expiration time is renewed plus 30 minutes from that time.

Body example:


```
{
    "authenticationMethod": {
        "userAuthentication": {
                "userEmail": "user@mail.com",
                "userPassword": "userPassword"
        }
    }
}
```

Reply example on success:

{

"resultCode": 1,

"resultMessage":"",

"sessionId":"8DBNVX7387RA7ZNSM7810NS"

}

Reply example on fail:

{

"resultCode": -1,

"resultMessage":"Error creating sessionId"

}

1.3.2 SessionId

This body example show how to use the sessionId created in the previous section. This body **is used before (see the example on section 1.4)** the other data fields.

Body example:

{

"authenticationMethod": {

"sessionAuthentication": {

"sessionId":"8DBNVX7387RA7ZNSM7810NS"

}

}

}

Reply example on success:

```
{
        "resultCode": 0,
        "resultMessage":""
}
```

Reply example on fail:

```
{
    "resultCode": -1,
    "resultMessage":"Error sessionId authentication"
}
```

1.4 Authentications methods & data fields

These body examples show how to use authentications methods with more body's data fields.

Body example with **userAuthentication**:

```
{
        "authenticationMethod": {
                "userAuthentication": {
                                "userEmail": "user@mail.com",
                                "userPassword": "userPassword"
        }
```

},

"sectionName": {

"key1": "value1"

}

}

Body example with **gToken**:

{

"authenticationMethod": {

"gToken": {

"token":"AABBCCDDEEFF99001122334455667788"

"vNodeId": 12345678912345678123

}

},

"sectionName": {

"key1”: "value1"

}

}

Body example with **sessionId**:

{

"authenticationMethod": {

"sessionAuthentication": {

"sessionId":"8DBNVX7387RA7ZNSM7810NS"

}
},
"sectionName": {
"key1": "value1"
}
}

2. Node Validations Methods

For some requests (Claim Node and Release Node) it is necessary validate the node. To do this validation there are two validation methods:

- **Registration Code Method** – For non-proprietary nodes of Domatica.
- **Label Key Method** - For proprietary nodes of Domatica.

2.1 Registration Code Method

The registration code can be obtained through proprietary Domatica software. This software is executed within the IoT device, thus generating its registration code, exclusively for that device.

Body example:

{
"nodeValidationMethod": {
"registrationCodeMethod": {
"regCode":
"7HDADNAG3477ABDDAB92BKJDASHDA73"

}

}

}

2.2 Label Key Method

Body Example:

{

"nodeValidationMethod": {

"labelKeyMethod": {

"labelKey": "8BSMAKF57BDS79VDD34W",

"serialNumber": 12349947773888292929292929

}

}

}

3. Management

3.1 Account management

3.1.1 Login

- Users login

- **Authentication method:** userAuthentication

Body example:


```
{
    "authenticationMethod": {
        "userAuthentication": {
            "userEmail": "user@mail.com",
            "userPassword": "userPassword"
        }
    }
}
```


Reply example on success:


```
{
    "resultCode": 0,
    "resultMessage":""
}
```

Reply example on fail:

{

"resultCode": -1,

"resultMessage": "You have entered an invalid userEmail or userPassword"

}

3.1.2 Register

- Users register (user information and entity information)
- **Authentication method:**

Body example:

{

"userInformation": {

"firstName": "user first name",

"lastName": "user last name",

"dateOfBirth": "02-19-1981",

"telephone": 222222222,

"userId": "userIddd",

"userEmail": "userId1@mail.com",

"userPassword": "pwd123#",

"typeOfUser":"admin"

},

}

"entityInformation": {

"entityId": "entity id",

"entityName": "entity name", "phoneNumber":

333333333,

"address": "entity address",

"city": "entity city",

"state": "entity state",

"zipCode": 64492,

"country": "entity country",

"accountContact": 84239,

"webSite": "waw.entityWebSite.pptt",

"vatID": 90692,

"activitySector": "entity activity sector"

}

}

Reply example on success:

{

"resultCode": 0,

"resultMessage":""

}

Reply example on fail:

{

"resultCode": -1,

"resultMessage":"Error user register"

}

3.2 Users management

3.2.1 New users

- Create new user
- **Authentication methods:** userAuthentication, sessionId

Body example:

{

"entityInformation": {

"entityId": "entity Id"

},

"userInformation": {

"firstName": "user first name",

"lastName": "user last name",

"dateOfBirth": "02-19-1981",

"telephone": 222222222,

"userId": "userId1",

"email": "userId1@mail.com",

"userPassword": "pwd123#"

}

```
}
```

Reply example on success:

```
{
    "resultCode": 0,
    "resultMessage":""
}
```

Reply example on fail:

```
{
    "resultCode": -1,
    "resultMessage":"Error adding new user"
}
```

3.2.2 Users information

- Get user information
- **Authentication method:** userAuthentication, sessionId

Body example:

```
{
    "entityInformation": {
        "entityId": "entityId"
    },
    "userInformation": {
```

"users": [{
"userId": "userId1"
},
{
"userId": "userId2"
}]
}
}

Reply example on success:

{
"userInformation": [{
"firstName": "user first name",
"lastName": "user last name",
"dateOfBirth": "02-19-1981",
"telephone": 222222222,
"userId": "userId1",
"email": "userId1@mail.com",
"userPassword": "pwd123#"
},
{
"firstName": "user first name",
"lastName": "user last name",
"dateOfBirth": "02-19-1981",

"telephone": 222222222,
"userId": "userId2",
"email": "userId2@mail.com",
"userPassword": "pwd123#"
}]
}

Reply example on fail:

{
"userInformation": []
}

3.2.3 Edit Users

- Edit user information
- **Authentication method:** userAuthentication, sessionId

Body example:

{
"userInformation": {
"userId":"userId1",
"newUserPassword": "newPwd123#",
"newEmail": "usernewEmail@email.com",
"newPhoneNumber": 9898989898
}
}

Reply example on success:

```
{
    "resultCode": 0,
    "resultMessage":""
}
```

Reply example on fail:

```
{
    "resultCode": -1,
    "resultMessage":"Error editing user"
}
```

3.2.4 Delete user

- Delete user(s)
- **Authentication method:** userAuthentication, sessionId

Body example:

```
{
        "userInformation": {
                "users": [{
                        "userId": "userId1"
                },
                {
```

"userId": "userId2"

}]

}

"entityInformation": {

"entity": "entity id"

}

}

Reply example on success:

{

"resultCode": 0,

"resultMessage":""

}

Reply example on fail:

{

"resultCode": -1,

"resultMessage":"Error deleting user"

}

3.3 Nodes management

3.3.1 Claim node

- Claim a node
- **Authentication method:** userAuthentication, sessionId

- **Node validation method:** registrationCodeMethod, labelKeyMethod • If the claim is successful we returns the vNodeId and serialNumber

Body example:

{

"entityInformation": {

"entityId": "entity Id"

},

"nodesInformation": {

"timeZone": "(GMT) Western Europe Time, London, Lisbon, Casablanca, Greenwich",

"geoLat": 38.7,

"geoLong": -9.14,

"client": "Client",

"tags": "Tags",

"name": "Name"

}

}

Reply example on success:

{

"resultCode": 0,

"resultMessage": "",
"vNodeId": "12345678912345678123",
"serialNumber": "837451839937122"
}

Reply example on fail:

{
"resultCode": -1,
"resultMessage": "Error claiming node″
"vNodeId": 0,
"serialNumber": 0
}

3.3.2 Release node

- Release a node(s)
- **Authentication method:** userAuthentication, sessionId

Body example:

{
"nodesInformation": {
"nodeId": [{
"vNodeId": 12345678912345678123
},
{
"vNodeId": 98765432198765432l123
}]

}

}

Reply example on success:

{

"resultCode": 0,

"resultMessage":""

}

Reply example on fail:

{

"resultCode": -1,

"resultMessage": "Error releasing node"

}

3.3.3 Replace node

- Replace a node
- **Authentication method:** userAuthentication, sessionId
- **Node validation method:** registrationCodeMethod, labelKeyMethod

Body example:

{

"nodesInformation": {

"oldSerialNumber": 9898445566888,

"vNodeId": 12345678912345678123

}

}

Reply example on success:

{

"resultCode": 0,

"resultMessage":""

}

Reply example on fail:

{

"resultCode": -1,

"resultMessage":"Error replacing node"

}

3.3.4 Node information

- List information about nodes(s) **Or**
- List information about nodes(s) that belong to user(s)

- **Authentication method:** userAuthentication, sessionId

Body example to **list information about node(s)** :

{
"queryMethod": {
"nodeId": [{
"vNodeId": "12345678912345678123"
},
{
"vNodeId": "987654321987654321123"
}]
},
"entityId": "entity id"
}

Body example to **list information about all nodes**:

{
"queryMethod": {
"nodeId": {
"vNodeId": " "
}
}
}

Reply example on success:

{

"nodesInformation": [{
"vNodeId": 12345678912345678123,
"phyId": 88982233,
"labelKey": "FFFF8888ZZZZ9999",
"timeZone": "(GMT) Western Europe Time, London,
Lisbon, Casablanca, Greenwich",
"geoLat": 38.7,
"geoLong": -9.14,
"client": "Client",
"tags": "Tags",
"name": "Name"
},
{
"vNodeId": 98765432198765432112З,
"phyId": 98984455,
"labelKey": "PPPP8888ZZZZ9999",
"timeZone": "(GMT) Western Europe Time, London,
Lisbon, Casablanca, Greenwich",
"geoLat": 38.7,
"geoLong": -9.14,
"client": "Client",
"tags": "Tags",
"name": "Name"

}]
}

Reply example on fail:

{
"nodesInformation": []
}

Body example to **list information about node(s) that belong to user(s)** :

{
"queryMethod": {
"users": [{
"userId": "userId1"
},
{
"userId": "userId2"
}]
},
"entityId": "entity id"
}

Reply example on success:

{
"nodesInformation": [{

"vNodeId": 999567812345678123,
"phyId": 88982233,
"labelKey": "FFFF8888ZZZZ9999",
"timeZone": "(GMT) Western Europe Time, London, Lisbon, Casablanca, Greenwich",
"geoLat": 38.7,
"geoLong": -9.14,
"client": "Client",
"tags": "Tags",
"name": "Name"
},

{
"vNodeId": 1234567812345678123,
"phyId": 98984455,
"labelKey": "PPPP8888ZZZZ9999",
"timeZone": "(GMT) Western Europe Time, London, Lisbon, Casablanca, Greenwich",
"geoLat": 38.7,
"geoLong": -9.14,
"client": "Client",
"tags": "Tags",

"name": "Name"

}]

}

Reply example on fail:

{

"nodesInformation": []

}

3.3.5 Edit node

- Edit information about specific node
- **Authentication method:** userAuthentication, sessionId

Body example:

{

"nodesInformation": {

"vNodeId": 1234567812345678123,

"newTimeZone": "(GMT) Western Europe Time, London,

Lisbon, Casablanca, Greenwich",

"newGeoLat": 38.7,

"newGeoLong": -9.14,

"newTags": "Tags",

"newName": "Name"

}

}

Reply example on success:

{

"resultCode": 1,

"resultMessage":""

}

Reply example on fail:

{

"resultCode": -1,

"resultMessage": "Error editing node"

}

3.4 Entity management

3.4.1 Entity information

- List entity information
- **Authentication method:** userAuthentication**,** sessionId

Body example:

{

"entityInformation": {

"entityId": "entityId",

}

}

Reply example on success:

```
{
    "entityInformation": [{
        "companyName":"entity name",
        "phoneNumber": 333333333,
        "address": "entity address",
        "city": "entity city",
        "state": "entity state",
        "zipCode": 64492,
        "country": "entity country",
        "accountContact": 84239,
        "webSite": "waw.entityWebSite.pptt",
        "vatID": 9069993232,
        "activitySector": "entity activity sector"
    }]
}
```

Reply example on fail:

```
{
    "EntityInformation": []
}
```

3.4.2 Edit entity

- Edit entity information

- **Authentication method:** userAuthentication, sessionId

Body example:

{

"entityInformation": {

"entityId": "entityId",

"vatID": 9069993232,

"newCompanyName": "new company name",

"newPhoneNumber": 111222333,

"newAddress": "new entity address",

"newState": "new entity state",

"newZipCode": 6666,

"newCountry": "new entity country",

"newAccountContact": 99999,

"newWebSite": "waw.newEntityWebsite.pptt"

}

}

Reply example on success:

{

"resultCode": 0,

"resultMessage":""

}

Reply example on fail:

```
{
    "resultCode": -1,
    "resultMessage": "Error editing entity!"
}
```

3.5 Tokens management

3.5.1 Add gToken

- Add a gToken to node
- **Authentication method:** userAuthentication, sessionId
- If the operation is successful we return the gToken

Body example:

```
{
    "tokenInformation": {
        "userId": "user1",
        "vNodeId": 1234567812345678123,
        "name": "token name"
    }
}
```

Reply example on success:

```
{
    "resultCode": 0,
    "resultMessage":"",
    "gToken": "11223344556677881122334455667788"
```

}

Reply example on fail:

```
{
    "resultCode": -1,
    "resultMessage": "Error adding gToken",
    "gToken": ""
}
```

3.5.2 Remove gToken

- Remove a gToken
- **Authentication methods:** userAuthentication, sessionId

Body example:

```
{
    "tokenInformation": {
        "token": "HDASI74732GHDJHDA63FFG2",
        "vNodeId": 1234567812345678123
    }
}
```

Reply example on success:

```
{
    "resultCode": 0,
```

"resultMessage":""

}

Reply example on fail:

{

"resultCode": -1,

"resultMessage": "Error removing gToken"

}

3.5.3 List my gTokens

- List all gToken by vNodeId and userId
- **Authentication methods:** userAuthentication**,** sessionId

Body example to list gToken(s) user associated to specific(s) nodes(s):

{

"nodesInformation": {

"nodeId": [{
"vNodeId": 12345678912345678123

},

{

"vNodeId": 987654321987654321123
}]
},

"userInformation": {

"userId": "user Id"

}

}

Body example to list all gToken(s) user associated to all nodes:

{

"nodesInformation": {

"nodeId": [{

"vNodeId": 0

}

},

"userInformation": {

"userId": "user Id"

}

}

Reply example on success:

{

"tokenInformation": [{

"vNodeId":1234567812345678123,

"gToken":

"11223344556677881122334455667788"

},

{

"vNodeId":1234567812345678123,

"gToken": "11223344556677881122334455667788"

}]

}

Reply example on fail:

{

"tokenInformation": []

}

4. Operation

4.1 Configuration

4.1.1 Deploy System Program

- Deploy new system program to a node
- **Authentication method:** userAuthentication, sessionId or gToken

Body example:

{

"vNodeId": 1234567812345678123,

"datatype": "base64XMLProgram",

"data": "YmFzZTY0IGlwZng=",

"maxWaitTime":300,

"parameters":{

}

}

Reply example on success:

{

"resultCode": 0,

"resultMessage": ""

}

Reply example on error:

{

"resultCode": -1,

"resultMessage": "error deploying system program"

}

4.1.2 Deploy user program

- Deploy new user program to a node
- **Authentication methods:** userAuthentication, sessionId or gToken

Body example:

{

"vNodeId": 1234567812345678123,

```
"datatype": "base64XMLProgram",
"data": "YmFzZTY0IHVzZXIgcHJvZ3JhbSB4bWw="
"maxWaitTime":300,
"parameters": {}
}
```

Reply example on success:

```
{
"resultCode": 0,
"resultMessage":""
}
```

Reply example on fail:

```
{
"resultCode": -1,
"resultMessage":"Error deploying user program"
}
```

4.2 Monitor and Control

4.2.1 Monitor – Request Log file

- Request log file
- **Authentication methods:** userAuthentication, sessionId and gToken

- Default tolerance: 15 minutes (900 seconds)

Body example requesting 150000 registers with tolerance of 5 minutes (300 seconds):

{

"vNodeId": 1234567812345678123,

"maxWaitTime": 300,

"parameters": {

"quantity": "150000",

"tolerance": "300"

}

}

Body example requesting by timestamp with tolerance of 5 minutes (300 seconds):

{

"vNodeId": 1234567812345678123123,

"maxWaitTime": 300,

"parameters": {

"beginTimestamp": "146376303",

"tolerance": "300"

}

}

Body example requesting between two timestamp periods with tolerance of 5 minutes (300 seconds):

{

"vNodeId": 1234567812345678123123,

"maxWaitTime": 300,
"parameters": {
"beginTimestamp": "146376303",
"endTimeStamp": "169377375",
"tolerance": "300"
}
}

Reply on success:

{
"resultCode": 0,
"resultMessage": "",
"nodesLog": [{
"timestamp": "121212121212",
"endPointId": "devId",
"endPointName": "idname",
"data": "2850"
},
{
"timestamp": "121212121212",
"endPointId": "devId",
"endPointName": "idname",
"data": "2850"

},

{

"timestamp": "121212121212",

"endPointId": "devId",

"endPointName": "idname",

"data": "2850"

}]

}

Reply on fail:

{

"resultCode": -1,

"resultMessage": "Error requesting log file",

"nodesLog": []

}

4.2.2 Monitor – Request System file

- Request system file
- **Authentication methods:** userAuthentication, sessionId or gToken

Body example:

{

"vNodeId": 1234567812345678123,

"parameters": {}

}

Reply on success:

```
{
    "resultCode": 0,
    "resultMessage": "",
    "systemFile": "YmFzZTY0Li4u"
}
```

Reply on fail:

```
{
    "resultCode": -1,
    "resultMessage":"Error requesting system file",
    "systemFile:""
}
```

4.2.3 Monitor - Request user program file

- Request user program file
- **Authentication methods:** userAuthentication, sessionId or gToken

Body example:

```
{
    "vNodeId": 1234567812345678123,
    "maxWaitTime":300,
    "parameters": {}
}
```

Reply on success:

```
{
    "resultCode": 0,
    "resultMessage": "",
    "userProgramFile": "YmFzZTY0Li4u"
}
```

Reply on fail:

```
{
    "resultCode": -1,
    "resultMessage":"Error requesting user program file"
    "resultFile":""
}
```

4.2.4 Monitor Control - Set

- Set multiple values across multiple nodes
- **Authentication methods:** userAuthentication, sessionId or gToken
- The yellow marked elements represent the vNodeId

Body example using gToken authenticaton and userAuthentication:

```
{
"nodes": {
    "121312312312": {
```

"authenticationMethod": {

"gToken": {

"token":

"AABBCCDDEEFF99001122334455667788",

"vNodeId": 12345678912345678123

}

},

"objects": {

"DCS_LEIRIA.CONTROL_ON_OFF": " "

}

},

"76826423424432423": {

"authenticationMethod": {

"userAuthentication": {

"userEmail": "user@mail.com",

"userPassword": "userPassword"

}

},

"objects": {

"DCS_LISBON.CONTROL_ON_OFF": " "

}

}

}

Reply example on success:

{

"nodes": {

"1234567812345678123": {

"result": "OK ",

"objects": {

"DCS_LEIRIA.CONTROL_ON_OFF": "1"

}

},

"1234605616430000000": {

"result": "OK ",

"objects": {

"DCS_LISBON.CONTROL_ON_OFF": "0"

}

}

}

}

Reply example on fail:

{

"nodes": {

"1234567812345678123": {

"result": "FAIL ",

"objects": {

"DCS_LEIRIA.CONTROL_ON_OFF": "1"

}

},

"1234605616430000000": {

"result": "FAIL ",

"objects": {

"DCS_LISBON.CONTROL_ON_OFF": "0"

}

}

}

}

4.2.5 Monitor Control - Get

- Get multiple values across multiples nodes
- **Authentications methods:** userAuthentication, sessionId or gToken
- The yellow marked elements represent the vNodeId

Body example using gToken authentication and userAuthentication:

{

"nodes": {

"121312312312": {

"authenticationMethod": {

"gToken": {

"token":

"AABBCCDDEEFF99001122334455667788",

"vNodeId": 12345678912345678123

}

},

"objects": {

"DCS_TEMPERATURE_LEIRIA.instShowValue": "

"

}

},

"7682642342432423": {

"authenticationMethod": {

"userAuthentication": {

"userEmail": "user@mail.com",

"userPassword": "userPassword"

}

},

"objects": {

"DCS_TEMPERATURE_LISBON.instShowValue": "

"

}

}

}

Reply example on success:

{
"nodes": {
"1234567812345678123": {
"result": "OK ",
"objects": {
"DCS_TEMPERATURE_LEIRIA.instShowValue":
"2650"
}
},
"1234605616430000000": {
"result": "OK ",
"objects": { "DCS_TEMPERATURE_LISBON.instShowValue":
"2550"
}
}
}
}

Reply example on fail:

{
"nodes": {

"1234567812345678123": {
"result": "FAIL ",
"objects": {
"DCS_TEMPERATURE_LEIRIA.instShowValue": " "
}
},
"1234605616430000000": {
"result": "FAIL ",
"objects": {
"DCS_TEMPERATURE_LISBON.instShowValue": " "
}
}
}
}