Universidade de Aveiro
2022

Joel Fernando
Bastos Baptista

Industrial data provider for consumption of Augmented Reality devices

Fornecedor de dados industriais para consumo por dispositivos de Realidade Aumentada

Universidade de Aveiro
2022

Joel Fernando Bastos Baptista

Industrial data provider for consumption of Augmented Reality devices

Fornecedor de dados industriais para consumo por dispositivos de Realidade Aumentada

Relatório de Estágio apresentado à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Mecânica, realizada sob orientação científica de Professor José Paulo Santos, Professor Auxiliar, do Departamento de Engenharia Mecânica da Universidade de Aveiro, e de Professor Paulo Miguel de Jesus Dias, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Esta dissertação teve o apoio dos projetos UIDB/00481/2020 e UIDP/00481/2020 - Fundação para a Ciência e a Tecnologia; e CENTRO-01-0145 FEDER-022083 - Programa Operacional Regional do Centro (Centro2020), através do Portugal 2020 e do Fundo Europeu de Desenvolvimento Regional.

Agradecimentos / Acknowledgements

Um especial agradecimento à minha família, com especial atenção ao meu pai e à minha mãe, Jorge e Denise, ao meu irmão Helder, à Rosana e ao Vilmar, que contribuiram de maneira única para o meu percurso académico e pessoal.
Agradeço também à Bosch Termotecnologia pela a oportunidade apresentada e à equipa AvP/MFD pela integração, companheirismo e ajuda no projeto, em especial ao meu supervisor Engenheiro Duarte Almeida.
Agradeço aos meus colegas de curso de Engenharia Mecânica que me acompanharam nesta jornada.
Agradeço, por último, à Universidade de Aveiro e aos orientadores, Prof. José Santos e Prof. Paulo Dias.

Keywords REST API, .NET Framework, Web Service, Industry 4.0, Apache Kafka Broker, Databases

Abstract

In response to the fourth industrial revolution, Bosch TT and the University of Aveiro (UA) started the Augmanity project, which aims to use innovative technologies such as Augmented Reality, IIoT, 5G, BigData and AI and Machine Learning, and apply them in the Portuguese business structure. This internship was inserted in the PPS4 subproject of the Augmanity project, specifically in the supply of data to be consumed by AR devices. The AR case studies will affect several areas of the company's value stream, from parts production machines to improving the efficiency of final lines, always with the aim of improving human performance. The complexity of this task comes from the fact that the necessary information comes from several different sources: the production line, Nexeed MES and SAP ERP, and the criterion that the information is accessed in real-time. The proposed system involves the creation of a web service that will provide all data available in a standardized way to be consumed by AR devices, like mobile phones, tablets and AR Glasses. For this, the developed system can access databases, Message Brokers and other web services and has a REST API architecture, developed in ASP.NET, to be able to transmit the data to the AR devices. In the end, industrial data provider was implemented on Bosch's internal network, managing to supply most of the data required by AR case studies. The web service underwent load tests to measure the wait times of the AR devices, obtaining satisfactory results and fulfilling the requirements of real-time communication.

Palavras-chave REST API, .NET Framework, Serviços Web, Industria 4.0, Apache Kafka Broker, Bases de Dados

Resumo

Em resposta à quarta revolução industrial, a Bosch TT e a Universidade de Aveiro (UA) iniciaram o projeto Augmanity que visa a utilizar tecnologias inovadoras como Realidade Aumentada, IIoT, 5G, BigData e AI and Machine Learning, e aplicá-las na estrutura empresarial portuguesa.
Este estágio inseriu-se no subprojecto PPS4 do projeto Augmanity, mais propriamente no fornecimento de dados para serem consumidos por dispositivos de RA. Os casos de estudo de RA irão afetar diversas áreas do fluxo de valor da empresa, desde máquinas de produção de peças, até à melhoria de eficiência de linhas finais, sempre com o intuito de melhorar a performance humana. A complexidade desta tarefa advém de a informação necessária ser proveniente de várias fontes diferentes: a linha de produção, o Nexeed MES e o SAP ERP, juntamente com o critério da informação ser acedida em tempo real. O sistema proposto envolve a criação de um serviço web que fornecerá todos os dados industriais de uma forma normalizada para serem consumidos pelos dispositivos de Realidade Aumentada, entre estes poderão ser telemóveis, tablets e AR Glasses. Para isso, o fornecedor de dados industriais desenvolvido é capaz de aceder a base de dados, Message Brokers e outros serviços web e possuí uma arquitetura de REST API, desenvolvida em ASP.NET, para conseguir transmitir os dados para os dispositivos de Realidade Aumentada.
No final, o sistema foi implementado na rede interna da Bosch, conseguindo devolver maioria dos dados requeridos pelos casos de estudo de RA. O serviço web sofreu testes de carga, para medir os tempos de espera dos dispositivos de RA, obtendo resultados satisfatórios e cumprindo os requerimentos de comunicação em tempo real.

Contents

List of Tables

Intentionally blank page.

List of Figures

Intentionally blank page.

Chapter 1

Introduction

1.1 Background and Motivation

The revolutionary step of Industry 4.0 heavily digitalized the production process. A direct consequence of this digitalization is the generation of large amounts of information related to the physical system, producing four challenges: data acquisition, transmission, processing, and visualization.

Augmented Reality (AR) is a new technology explored to answer the data visualization problem. AR is the concept of adding virtual components to an environment. Its display options are Hand-Held Displays (HHD), like smartphones, Spatial Displays like digital projectors, or Head-Mounted Displays, like smart glasses. Although AR technology is relatively new to the industrial world, it is expected to grow significantly alongside Virtual Reality in maintenance, decision-making, and training [10]. To explore this growing potential, Bosch TT proposed three AR use cases. These use cases belong to the Augmanity project, where Bosch TT and the University of Aveiro started exploring innovative technologies. The study cases are related to the PPS4 – Artificial Vision and Augmented Reality.

- **Use case 1 - Equipment Technical Data**: Display real-time data associated with the equipment being analyzed in AR to help the maintenance department.

- **Use case 2 - Assisted Production**: Helps the operator by showing which parts are necessary for assembling different kits. This will reduce training costs and significantly decrease this sector's mistakes.

- **Use case 3 - Final Assembly Line - Real-Time Monitoring**: A production line with 10 to 20 stations needs to allow users with AR equipment to consume the information relative to the station in the visual field. This will help the managers of the production line to make a faster analysis of every significant data of the production process.

The problem facing Bosch's use cases is relative to information transfer. Almost every case study requires real-time data from some part of the company, which imposes a barrier that needs to be crossed for implementing AR solutions.

1.2 Aim

This project aims to create a uniformised industrial data provider for AR devices. This data transfer will be accomplished by a system that retrieves information throughout the Plant and possesses a Web API module that allows web communication between the system and the AR devices. The solution needs to be capable of responding to the use cases' requirements in terms of data and real-time criteria. The use cases' requirements are described in the following subsections.

1.2.1 Use case 1 - Equipment Technical Data

This case study aims to aid the maintenance personnel in the stamping press Haulick & Roos RVD200T, shown in figure 1.1. This press possesses 25 sensors, retrieving data about temperature, electric tension, motor speed, oil pressure, and tool displacement and vibration. The idea is to provide the sensor information, technical drawings of the equipment, recorded maintenance, and troubleshooting guides to facilitate the maintenance process of the machine. The challenge in this use case is the disparity of the data provided. The system must be capable of acquiring information from the production line, SAP ERP, and local folders that store the videos and documents.

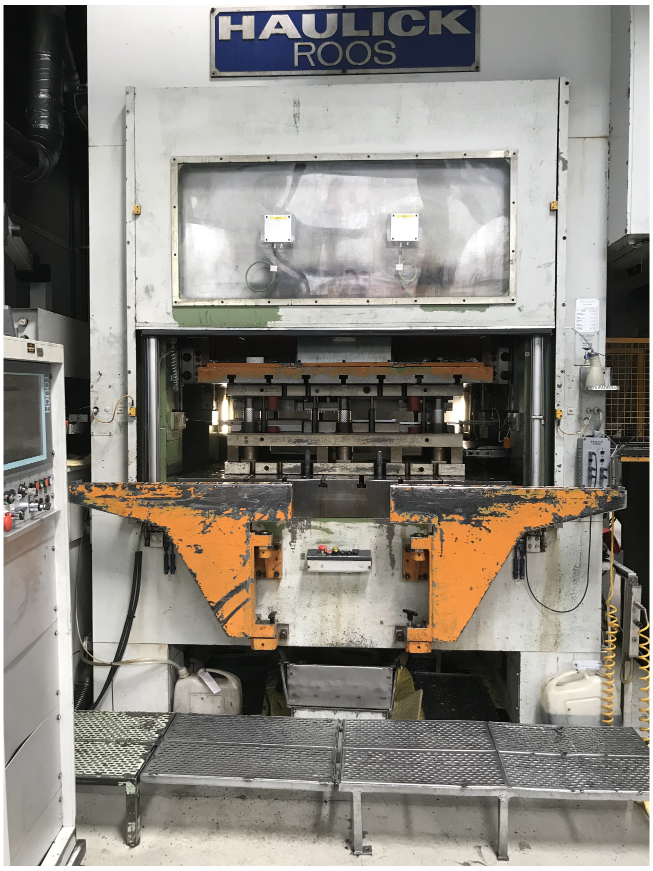


Figure 1.1: Bosch's Haulick & Roos RVD200T Stamping Press

1.2.2 Use case 2 - Assisted Production

This case study aims to enhance line S854 by implementing AR devices to aid the workers. This line focuses on producing kits composed of parts and materials sold alongside the boilers and water heaters. In this case, SAP ERP possesses all the relevant information.

1.2.3 Use case 3 - Final Assembly Line - Real-Time Monitoring

This case study has the goal of helping the line manager of final assembly lines 7 and 10. An example of final assembly lines is shown in figure 1.2. The lines have, as inputs, all the manufactured parts necessary for the boilers and water heaters. Then, each station performs processes of assembly or testing until the product is packed and ready for shipping. The manager's task is to track KPIs, like OEE, FPY, Productivity, and cycle times, to make quick adjustments to the line and ensure the overall success of the current shift. The idea is to facilitate this task by displaying this information using AR devices. In this case, the Neexed MES calculates and stores the KPIs of the mentioned lines and the respective stations.

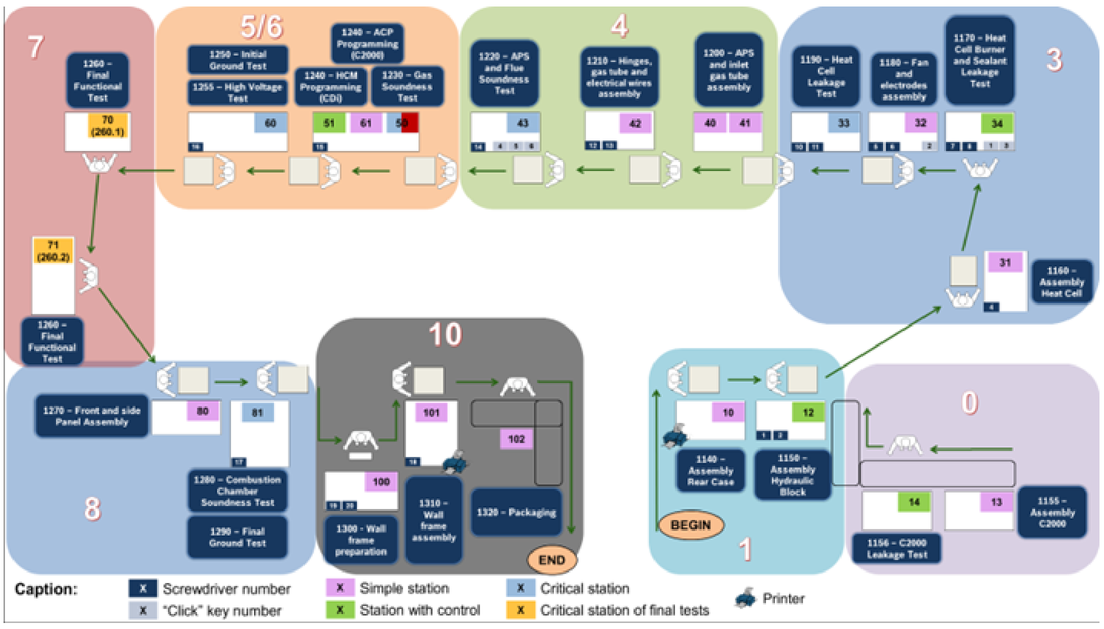

7
1260 – Final Functional Test
70 (260.1)
71 (260.2)
1260 – Final Functional Test
5/6
1250 – Initial Ground Test
1255 – High Voltage Test
1240 – ACP Programming (C2000)
1240 – HCM Programming (CDI)
1230 – Gas Soundness Test
60
51
61
50
4
1220 – APS and Flue Soundness Test
1210 – Hinges, gas tube and electrical wires assembly
1200 – APS and inlet gas tube assembly
43
42
40
41
3
1190 – Heat Cell Leakage Test
1180 – Fan and electrodes assembly
1170 – Heat Cell Burner and Sealant Leakage Test
33
32
34
31
1160 – Assembly Heat Cell
8
1270 – Front and side Panel Assembly
80
81
1280 – Combustion Chamber Soundness Test
1290 – Final Ground Test
10
101
102
100
1300 - Wall frame preparation
1310 - Wall frame assembly
1320 – Packaging
END
1
10
12
1140 – Assembly Rear Case
1150 – Assembly Hydraulic Block
BEGIN
0
14
13
1155 – Assembly C2000
1156 – C2000 Leakage Test
Caption:
X Screwdriver number
X "Click" key number
X Simple station
X Station with control
X Critical station
X Critical station of final tests
Printer

Figure 1.2: Final Assembly Line

1.3 Document Organization

The present document is divided into five topics in the following order:

- **State of the Art** - A review of theoretical concepts used in the realisation of the project. These concepts are Industry 4.0, Web Services, Frameworks, Message Brokers, and Nexeed MES.
- **Proposed Solution** - Presents the conceptual architecture of the communication system and how it aims to fulfil all the requirements imposed by the PPS4 use cases.
- **Implementation** - Presents the practical architecture of the communication system and how it was implemented inside Bosch's environment.
- **Test Solution** - Methods used to evaluate the implemented system alongside analysing the results.
- **Conclusion** - Conclusion about the implemented system and suggestions for future projects.

Chapter 2

State of the Art

2.1 Industry 4.0

Industry 4.0 can simply be put as the transformation from machine dominant manufacturing to digital manufacturing [11]. In more complex terms, Industry 4.0 is defined as the advanced digitalization of factories using Internet technologies and future-oriented technologies. This paradigm shift introduces the production process to modular and efficient manufacturing systems and scenarios where the products control their production process [12].

Several design principles and technology trends can be identified in Industry 4.0. These design principles focus primarily on [13] horizontal and vertical integration, decentralization, interoperability, modularity, product and service individualization and real-time capability.

These principles usually are implemented using new technologies. There are a wide variety of technologies being used throughout the industrial world, with the following being the most prominent [13]:

- Augmented and virtual reality
- Industrial internet of things
- Automation and industrial robotics
- Cyber-physical systems
- Cybersecurity
- Cloud and data computing

2.1.1 Industrial Internet of Things

Industrial Internet of Things (IIoT) is using the Internet of Things (IoT) technologies for manufacturing purposes. More concretely, IIoT is a system that consists of networked smart objects, cyber-physical assets, associated generic information technologies, and optional cloud or edge computing platforms. This combination enables real-time, intelligent, and autonomous access, collection, analysis, communications, and exchange of process, product, or service information in the industrial environment to maximize production value. This value is an example of improving product or service delivery, increasing productivity, lowering labour costs, lowering energy usage, and shortening the build-to-order cycle [14].

This concept creates another notion called Industrial Internet. The primary necessity to create this concept was to differentiate it from the consumer/social internet and their value creation cores. Industrial Internet has two main focuses. The first is connecting the machine sensors and actuators to the local processors and internet, and the other is connecting different industrial networks [13].

2.1.2 Cyber-Physical Systems

Cyber-Physical System (CPS) is a concept associated with Smart Manufacturing. A CPS integrates communication and control between the cyber and the physical world in real-time while enhancing the transparency in the process [15] [1]. Figure 2.1 shows the information transactions in CPS. A Cyber-Physical Production System is a CPS designed to aid the production process and usually involves communication between the sensors and actuators and the decision-making software [16].

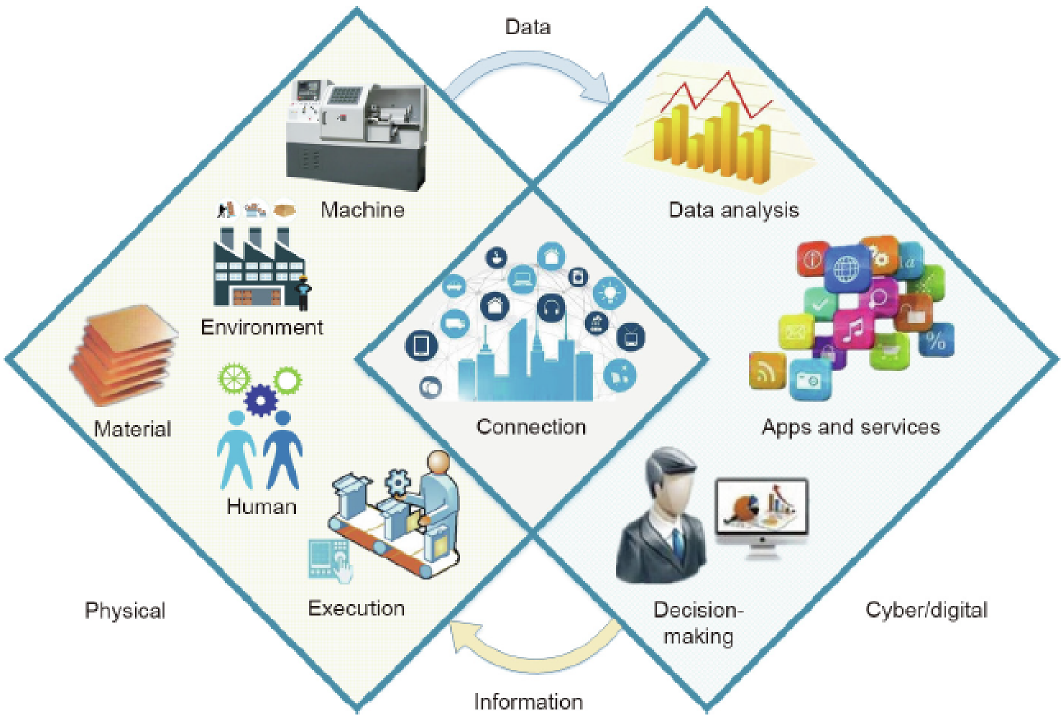


Figure 2.1: Mapping information between the physical and cyber world [1]

2.1.3 Industrial Augmented Reality

Industrial Augmented Reality (IAR) uses Augmented Reality (AR) technologies to support or enhance the industrial process. The concept of AR is applied to every environment where virtual components have been added or replaced by a physical component [10].

IAR is expected to change the way operators perform their daily tasks by equipping them with interfaces that provide information generated from the production process or Decision Support Systems (DSS). The IAR concept is being used to assist workers in maintenance, remote assistance, decision making and training [17].

2.1.4 ERP

ERP (Enterprise Resource Planning) systems are the business's backbone, allowing them to manage all organizational resources and transactions through a single system. ERP systems are standardized, off-the-shelf software solutions based on industry best practices [18]. Contrasting with MES, ERP systems tend to have a more long-term aspect in their decision-making, controlling a wide range of operations, such as logistics, transportation management, finance, material use, shipping, customer relationship management, and human resources.

2.2 Nexeed MES

Production and maintenance techniques underwent a significant change during the fourth industrial revolution. One of these changes was the implementation of Manufacturing Execution Systems (MES). These systems enhance large companies' productivity, quality, and agility, specifically those that seek to establish trade networks worldwide [19]. These systems achieve this by monitoring production and controlling the company's tasks [20], making data available in real-time, thus helping decision-making. MES is also helpful by being a link between ERP systems and the shopfloor [21]. The integration of MES in the automation hierarchy is shown in figure 2.2

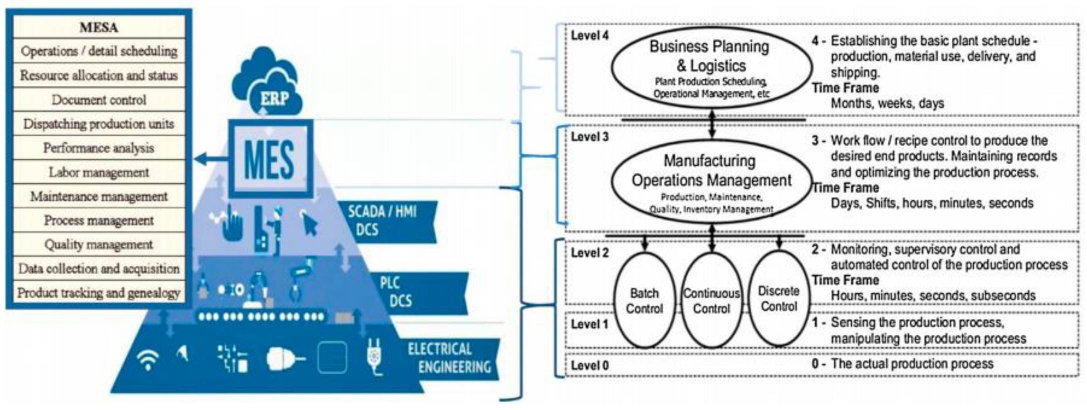

MESA
Operations / detail scheduling
Resource allocation and status
Document control
Dispatching production units
Performance analysis
Labor management
Maintenance management
Process management
Quality management
Data collection and acquisition
Product tracking and genealogy
ERP
MES
SCADA / HMI DCS
PLC DCS
ELECTRICAL ENGINEERING
Level 4
Business Planning & Logistics
Plant Production Scheduling, Operational Management, etc
4 - Establishing the basic plant schedule - production, material use, delivery, and shipping.
Time Frame
Months, weeks, days
Level 3
Manufacturing Operations Management
Production, Maintenance, Quality, Inventory Management
3 - Work flow / recipe control to produce the desired end products. Maintaining records and optimizing the production process.
Time Frame
Days, Shifts, hours, minutes, seconds
Level 2
Batch Control
Continuous Control
Discrete Control
2 - Monitoring, supervisory control and automated control of the production process
Time Frame
Hours, minutes, seconds, subseconds
Level 1
1 - Sensing the production process, manipulating the production process
Level 0
0 - The actual production process

Figure 2.2: MES in Automation Hierarchy [2]

The ANSI/ISA-95 norm attributes twelve functions every MES should aim to achieve. These required functions are [3]:

- Resource Allocation and Control
- Dispatching production
- Data collection and acquisition
- Quality operations management
- Process management
- Production tracking
- Performance analysis
- Operations and detailed scheduling
- Document control
- Labour management
- Maintenance operations management
- Movement, storage, and tracking of materials

Nexeed MES is a system implemented in Bosch that aims to fulfil all the MES requirements. This system is modular, with each module fulfilling a specific role. Figure 2.3 shows the Nexeed MES modules [22].

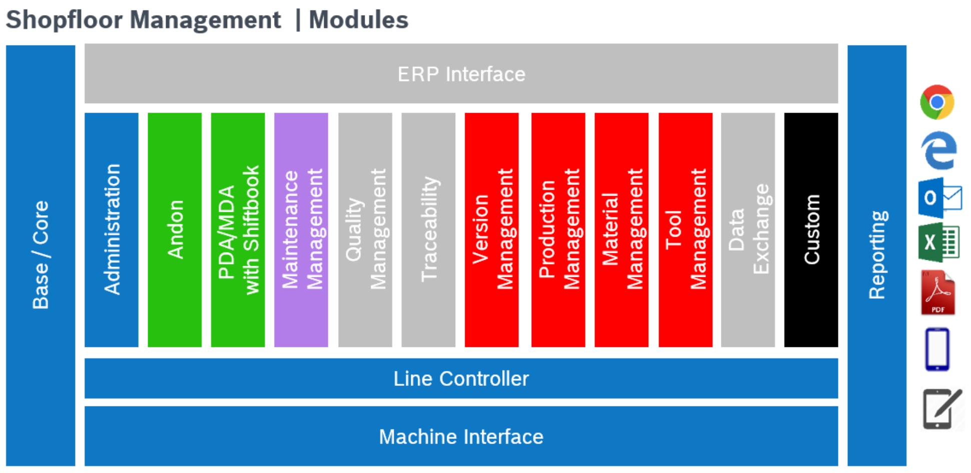


Figure 2.3: Shopfloor Management Modules of Nexeed MES [3]

- **Administration** – It creates the user accounts and manages their roles and permissions.
- **Line Controller** – It encapsulates the vMDT, which manages the processing order of the parts.
- **Machine Interface** – It connects the Nexeed MES modules with the production lines' machines.

- **Reporting** – It generates various reports for the OIS.NET portal.
- **Adon** – It allows machine operators to view important production indicators in real-time.
- **PDA/MDA** – It generates helpful information regarding the production, like Key Performance Indicators (KPI), to be consulted in real-time.
- **Shiftbook** – It exposes information about the work shifts, production planning and shift stops.
- **Quality Management and Traceability** – It presents data regarding traceability and quality of the production process through the production lines.
- **Data Exchange** – It transfers data between modules.
- **Version Control** – Keeps the machines' software updated.
- **Setup Control** – It allows editing production recipes.
- **Material Control** – It introduces traceability of the materials through the production stream.
- **Tool Control** – It controls the usage of the tools and implements predictive maintenance.
- **Maintenance Management** – It aims to optimize the maintenance process, allowing the technicians to do the maximum operations possible remotely.

The Nexeed MES has three independent servers: Application Server, Database Server, and Web Server. The first server is responsible for the data collection and processing of the shop floor and contains the previously mentioned modules. The second server is the database that stores all the production data and is accessible to the other two servers. Lastly, the Web Server is an interface to the user and can be used in the browser in the OIS.NET portal. Figure 2.4 illustrate the systems architecture.

2.3 Web Services

A web service is a way of transferring information between devices in a network. Many models implement this communication, but modern technology uses SOAP or REST [23]. This service usually does not provide a user GUI and has the primary function of data exchange between programs. These applications do not need to be compatible with the language or platform [24].

2.3.1 Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) uses HTTP and XML to communicate between nodes. There are three main types of SOAP nodes: senders, receivers, and intermediaries. The SOAP sender generates and transmits the message, while the receiver can receive and process it. The third node has the two previous functionalities and serves to redirect the message, so it arrives at the last SOAP receiver node [24]. The message

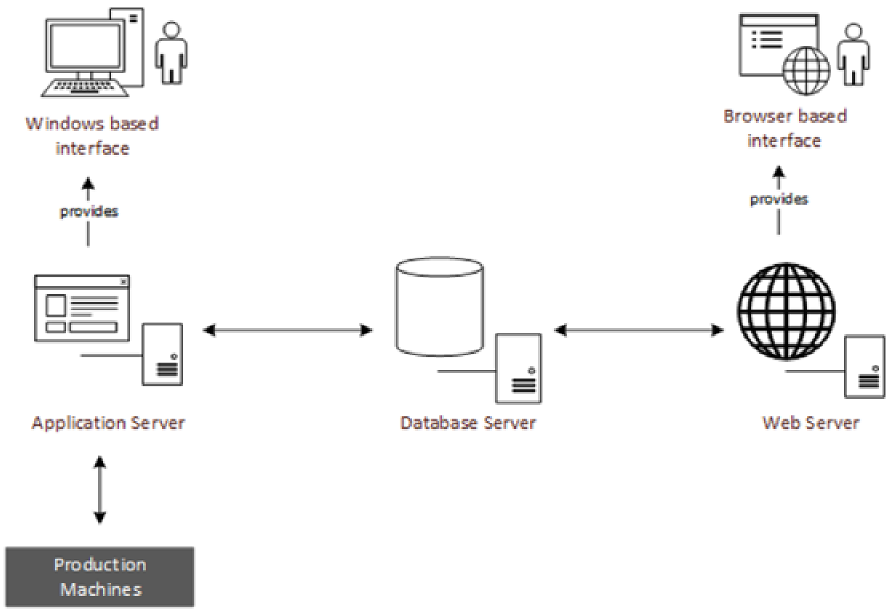


Figure 2.4: Nexeed MES Architecture [3]

structure of SOAP is an envelope with two fields: header and body. The header has information relative to the application's data types and authentications. For this reason, the header field may be optional. However, the mandatory body field contains the information that must reach the client application [24] [25].

2.3.2 Representational State Transfer (REST)

REST stands for Representational State Transfer, an architectural style for distributed hypermedia systems. Roy Fielding was the first to propose this architectural style in his PhD dissertation, imposing some constraints on web communication. These constraints belong to six different categories [26] [4] [27]:

1. Client-server
2. Uniform Interface
3. Layered system
4. Cache
5. Stateless
6. Code-on-demand

The communication is Client-server based, separating the consumer and producer of the information, allowing them to have independent implementations.

The Uniform Interface has four constraints to allow all users to communicate without breaking the system.

1. Identification of resources: The web resources need to be uniquely by a Uniform Resource Identifier (URI).
2. Manipulation of resources through representations: Clients control how the resources representation, allowing data consumption from different types of programs.
3. Self-descriptive messages: Clients can send a message with a resource's desired state, and the server can respond with a message with the resource's current state. The self-descriptive message usually sends metadata with additional details regarding the resource state, the representation format and size, and the message itself. An HTTP message provides headers to organize the various types of metadata into uniform fields.
4. Hypermedia as the engine of application state (HATEOS): A resource's state representation includes links to related resources.

The layered system constraints allow the existence of proxies and gateways as intermediaries transparently to the server and client. These usually are used to enforce security, response caching, and load balancing.

The cache constraint instructs a web server to declare the cacheability of each response's data, improving the communication performance by responding to a new request with saved responses. This met can help reduce client-perceived latency, increase overall availability and reliability, and control a web server's load.

The web server does not require to store the state of the client applications, which means that the client applications must include all the relevant information in each interaction with the server.

2.3.3 REST vs SOAP

Overall, there are some critical differences between communication technologies. Firstly, REST is more lightweight and requires less bandwidth than SOAP. Additionally, the REST's message type is not defined, allowing it to be JSON or plain text. In addition, SOAP needs to send messages with XML, making it less versatile. These differences make REST a more friendly option to implement in wireless communications [23]. Finally, REST leads to more scalable, safe, effective, and reliable solutions, and for these reasons, it has become a great alternative to SOAP [28].

2.3.4 REST API

REST API is a concept born by joining two existing concepts. First, there is the concept of REST, explained in subsection 2.3.2. Then, there is the API concept. An Application Programming Interface (API) is a middleware that maps services' functions and data sets for easy and controlled access to client applications. When an API serves as an interface to a web service, it becomes a web API, and if it obeys the REST architecture, it is called a REST API. Typically, these REST APIs communicate via HTTP protocol and can implement layers of security to the service that is being exposed [4].

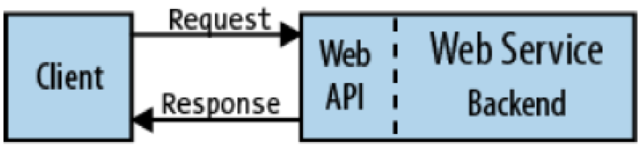


Figure 2.5: Web API generic application [4]

The definition of REST API does not propose an implementation architecture, and it only has the limitations of the REST constraints and API functions. However, the Model-View-Controller (MVC) architecture is usually the choice for interface applications [29]. Figure 2.6 shows this design pattern that separates the application into three main modules: Model, View and Controller [5].

- Model – The Model represents the data structures used by the other two modules.
- View – The View controls how a user will consume the data.
- Controller – The controller receives the user's commands and transforms them into actions for the other two models.

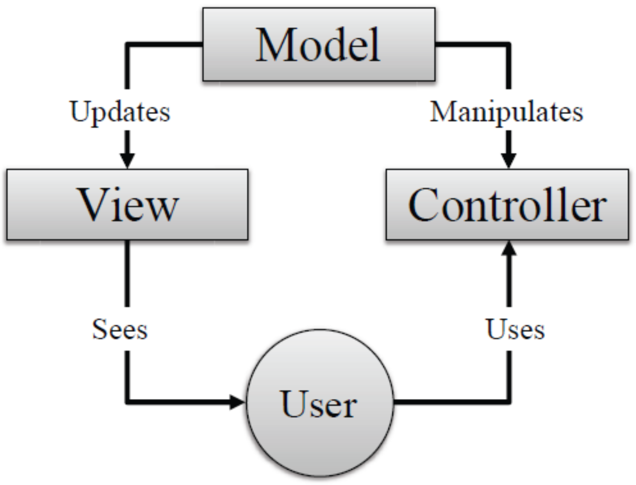


Figure 2.6: Model-View-Controller Architecture [5]

This concept lacks scalability, but with slight modifications, it changes into a Hierarchical-Model-View-Controller (HMVC). HMVC architecture can be summarized into interactive MVC sub-structures, mainly with the Controllers communicating with each other, as shown in figure 2.7.

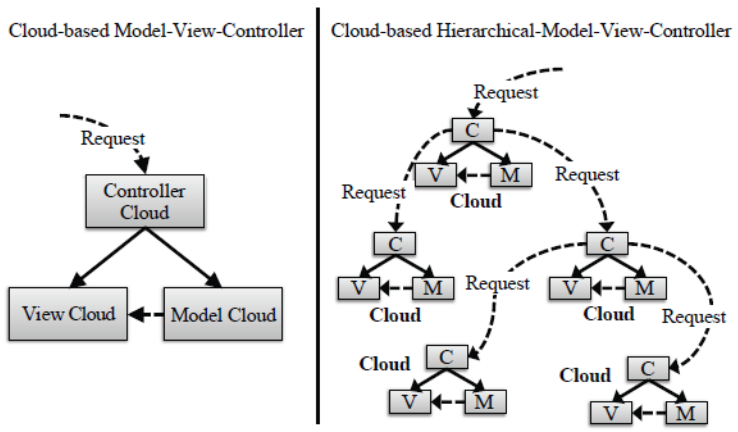


Figure 2.7: Differences between MVC and HMVC Architectures [5]

2.4 Frameworks

A Framework is a collection of reusable and standard components that produce an architecture with a specific behaviour [30]. These frameworks aim to create new projects that inherit the Framework's low-level functionalities, allowing the developer to add new high-level functionalities to solve the new problem. There are many advantages to using existing frameworks in the developing environment. Firstly, it allows the creation of a viable solution much faster because a portion of the project's code is already developed and tested. Secondly, uniformly structuring a project can lead to better coding practices. Lastly, Frameworks allow for improving the solution performance and quality without additional effort for the programmer [31].

Many open-source frameworks are available, with many software architectures and design patterns implemented in their templates. Typically, if the objective is to produce software with similar behaviour to existing solutions, the chances are that there is a Framework that provides a template for it. In developing web services, some of the existing Frameworks that provide solutions are .NET Framework and Flask Framework.

2.4.1 .NET Framework

The .NET Framework was initially released by Microsoft in 2002 and received constant improvement and support throughout the years, becoming a standard and valuable tool for software development [6]. Initially, the .NET Framework consisted of the common language runtime (CLR) and the Framework Class Library (FCL) [32]. The CLR's objective is to manage, load and locate .NET objects and manage low-level tasks like memory management, threads coordination, hosting and security. Incapsulated in CLR, there are the Common Type System (CTS) and the Common Language Specification (CLS). The CTS specifies the data types and programming constructs supported by the CLR and how they interact. Lastly, the CLS defines common types and programming constructs of .NET. The FCL, also known as Base Class Library (BCL), provides functions that the project's classes can inherit [33] [6]. Figure 2.8 shows the relation between BCL, CLR, CTS and CLS.

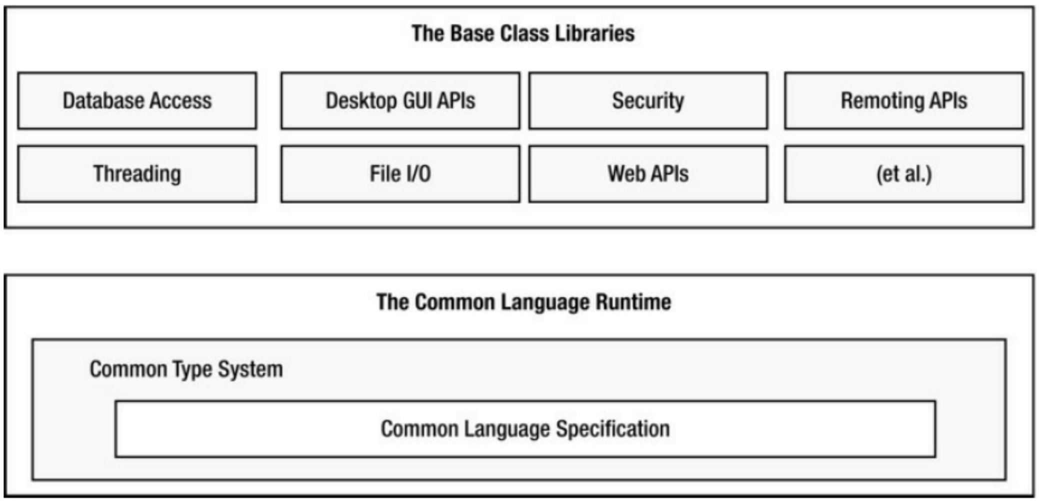


Figure 2.8: Relation between BCL, CLR, CTS and CLS [6]

The .NET Framework allows doing different projects, like Web Services, Web Forms and Windows Forms. For this project, the exciting project type is the Web Service. This Framework has a template called web api that implements a Web Service with web api functionalities and an MVC pattern. This template supports different language types, including C#.

2.4.2 Flask Framework

Framework Flask is a Python-based web framework. Flask is a library with the primary purpose of creating web applications. The objective of this Framework is to be lighter and depend less on external libraries [34]. Because of its simplicity, Flask is considered a microframework with few functionalities. However, it is possible to add new libraries to Flask to attribute new features needed for the designed web service.

Flask is composed of two libraries: Werkzeug and Jinja2. The first library deals with web communication by assuring the routing and Web Server Gateway Interface and adding debugging options. The second module is a template engine, which can produce desired output formats like HTML to create user interfaces [35].

2.5 Databases

A database's primary function is to allow one or more applications to store and retrieve information conveniently and efficiently. Usually, databases are used to manage large amounts of data, so they need to be carefully designed to respond to the user's requirements without losing efficiency [7]. Typically, a database is used in vertical architecture, as shown in figure 2.9.

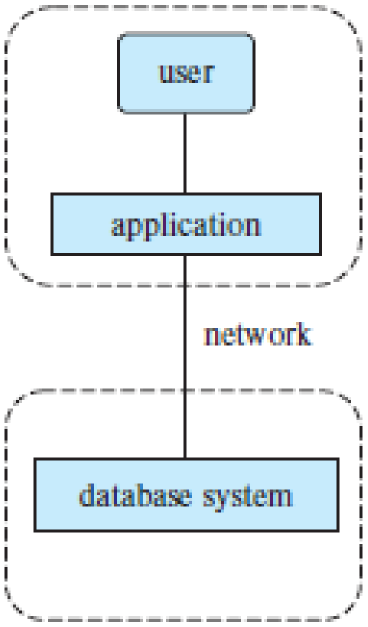


Figure 2.9: Generic system implementation with a database [7]

A database is mainly a collection of tables. A table represents the information that needs to be stored, with each table representing a different object or concept. A table has columns and rows. Each row is a record or an entry in the table, and each column is a property or attribute of the object being modelled by the table [36].

However, databases needed software to manage them, so database management systems (DBMS) were created. These systems are responsible for implementing the behaviours that allow applications to access the databases.

2.6 Message Broker

A Message Broker (MB) is a concept created to solve a problem of distributed system and establish communication between multiple isolated processes. The Message Broker, also known as a messaging queue, is usually used in Streaming Processing Engines (SPE) with the goal of real-time processing data to solve existing problems. A MB is helpful when there is more than one source of data and if these sources do not output persistent data [8]. Figure 2.10 shows an example of a data streaming system that implements a message broker.

Figure2.10 shows that the architecture possesses various sources, with different message types, sending data to the message broker. The message broker uniformizes the data stream and allows one consumer to read the data in a controlled manner. In addition, a message broker should allow multiple consumers to consume the data stream [8].

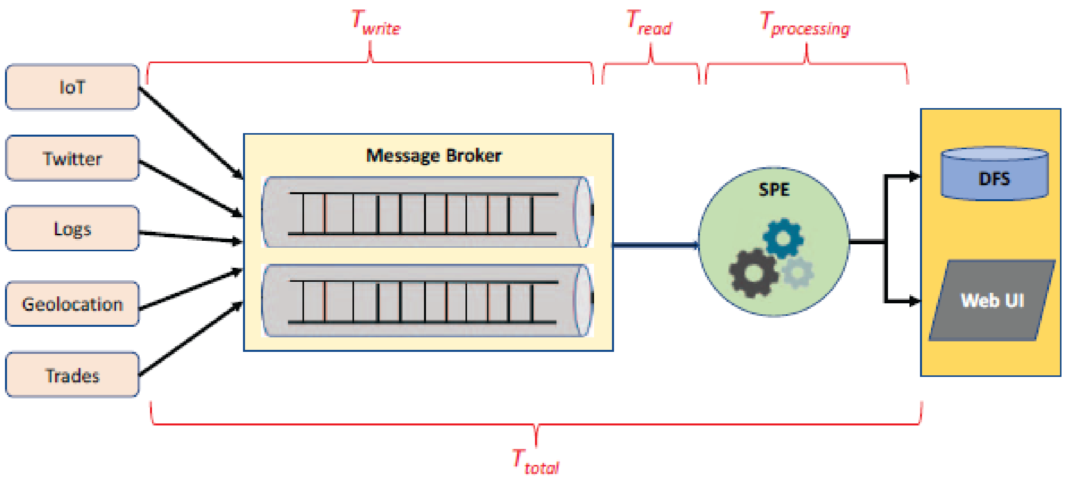


Figure 2.10: Generic Stream Processing Pipeline [8]

2.7 Apache Kafka

Apache Kafka is a framework that allows system integration, granting it capability of integrating Messaging Broker systems. These systems allow seamless integration of information pipelines from producers to consumers, keeping them separate through the process. Kafka supports data transfers of millions of messages per second. It is compatible with multiple clients from different platforms, like Java, .NET, PHP, Ruby, and Python, making it a viable system for big data real-time solutions [37].

Kafka is a commit log service that is distributed, partitioned, and replicated. The system achieves this by maintaining feeds of messages that are called topics. In these topics, producers can publish messages, and consumers can read them. Each broker can have multiple topics, and each implementation can have multiple brokers, creating a broker cluster. Lastly, topics are divided into partitions, ordered, immutable message sequences [9]. Figure 2.11 presents a generic architecture of an Apache Kafka Broker implementation.

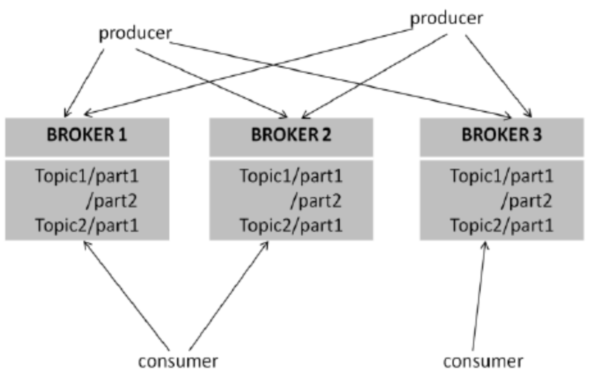


Figure 2.11: Apache Kafka architecture [9]

Chapter 3

Proposed Solution

As mentioned in the Introduction Chapter, the project's goal is to create a web service that gathers information from the plant to satisfy the objectives of the Augmented Reality implementations. The proposed solution in figure 3.1 aims to fulfil the requirements of the three AR use cases. The idea is to implement a web service with REST API architecture that communicates with several of Bosch's programs and exposes the information to the AR devices wirelessly, using a server-client communication type. Each data source serves a purpose for at least one use case.

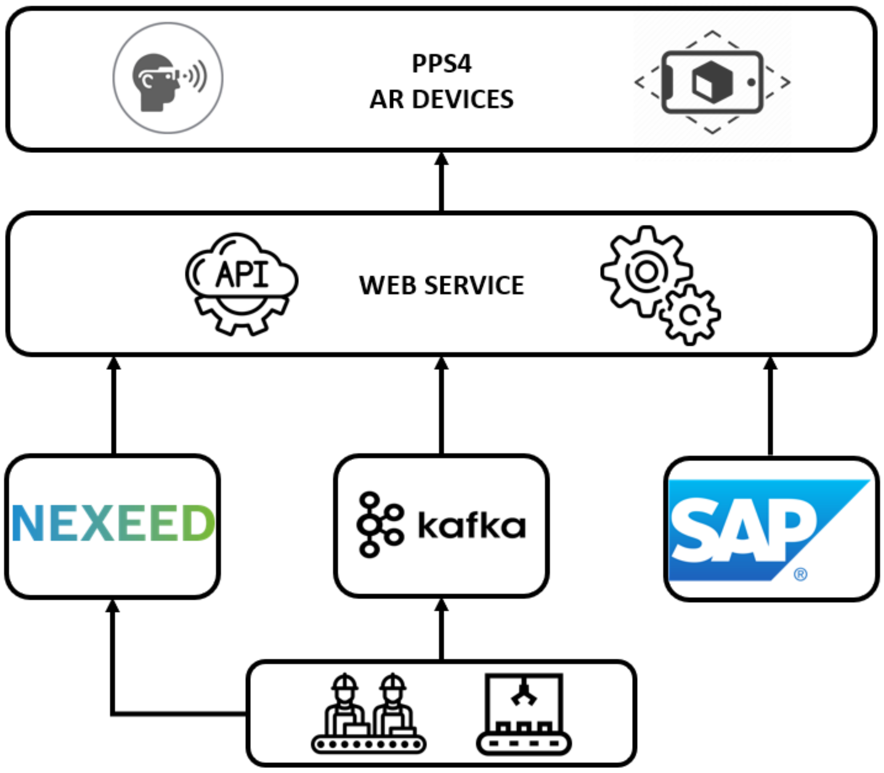


Figure 3.1: Proposed Solution architecture

3.1 SAP ERP

SAP ERP, as a resource management system, possesses a lot of crucial company information. This system is essential for use case 1 – Equipment Technical Data because it stores information about the maintenance records and the tools of the Haulick & RVD200T Press. SAP ERP also manages the work orders and materials lists of the kits of line S854, which is present in use case 2 – Assisted Production. The work orders are Kanban cards that identify the quantities of kits for production and the kit's materials and location.

3.2 Nexeed MES

The system needs to be able to communicate with Nexeed MES for use case 3 – Final Assembly Line – Real-Time Monitoring. This use case requires making accessible the KPIs of lines 7 and 10 for their stations. The Nexeed MES system takes input data from the production lines and manages them. It also calculates Key Performance Indicators (KPI), which can be accessible by third-party software. The KPIs necessary are Overall Equipment Efficiency (OEE), First Pass Yield (FPY), Productivity, and the number of parts processed (partCount). These are the most common KPIs analyzed by the line manager to administrate changes to the current shift.

3.3 Production Line

The system needs to provide data about the sensors implemented in the stamping press in use case 1 – Technical Equipment Data. Alongside this, the primary requirement is for the data to be in real-time, so the proposed solution aims to retrieve data directly from the production line. The web service must also get information from lines 7 and 10 to calculate the cycle times of the stations to fulfil the requirements of use case 3.

The real-time data consumption is done by listening to topics on a messaging broker implemented in Bosch. This messaging broker possesses topics that represent every digitalized line in the company. Some services stay listening for XML telegrams sent from the production line. These services map the telegram's contents to a JSON message with similar fields and publish the result to the respective topic. These topics are available to third-party software to consume.

3.4 REST API

The main module of the web service is a REST API, which will function as a web service. The reasons to use a REST API as a tool to send data to other software are listed below.

Firstly, the REST architecture standardizes communication by implementing the protocols already used by the World Wide Web. This factor is crucial as this system needs to be consumed by Augmanity project collaborators, who will be outside the Bosch Termotechnology network. However, if it were not for this restriction, having a system prepared to communicate over an internet network meets the goals of Industry 4.0, which is standard for companies today.

Secondly, as the REST API is an interface, the way the customer interacts with the plant's information is entirely controlled, adding a certain level of security. This imposition means that the client can only obtain the data it is supposed to consume, as it does not have direct access to databases or local folders hosted on the company's internal servers.

Finally, the modular architecture of the REST API allows for expanding the system in the future without compromising its previous functions. As the REST API controllers work independently, creating, changing, or removing a controller does not affect the system's proper functioning. This property allows for the system's continuous development so that one day it will stop being a service for the consumption of Augmented Reality devices and becomes a web service of the company, capable of providing any data present on the plant premises.

Intentionally blank page.

Chapter 4

Implementation

The implemented system aimed to fulfil all the requirements imposed in the Introduction. Figure 4.1 presents the implemented web service's architecture, a more practical approach to the proposed solution architecture in figure 3.1.

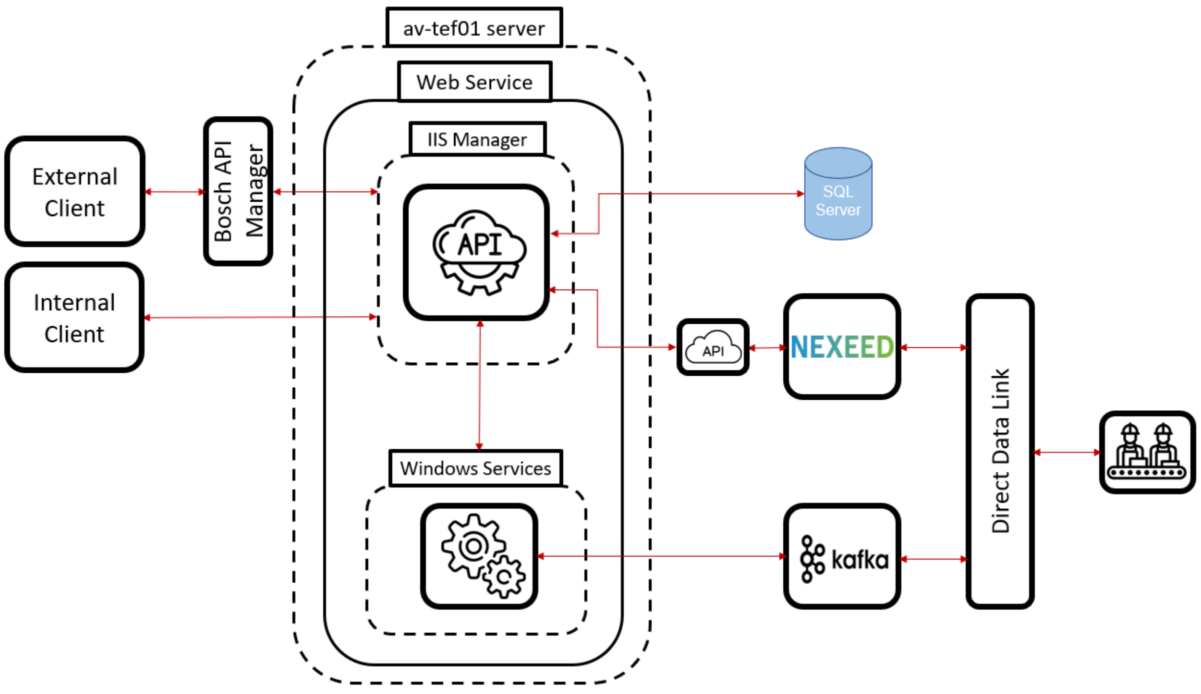


Figure 4.1: Implemented System Architecture

Figure 4.1 needs to be analyzed from right to left. On the right side of the architecture, three data sources can be seen: SQL server, Nexeed MES, and Production Line. These three data sources create three pipelines that the system will handle.

Firstly, there is the Production Line pipeline. In Bosch, there is already established DirectDataLink, a system to redirect the XML messages from the Production Line to Nexeed MES. However, this system duplicates the information and sends it to an Apache Kafka Broker. This Message Broker is consumed by the *KafkaConsumer* module of the system, which saves a small amount of data for the REST API module access whenever necessary.

Secondly, there is the Nexeed MES pipeline. In Bosch, there are APIs for Nexeed MES that mimic the stored procedures inside the Nexeed MES database. The system's REST API module requests these APIs whenever it is needed.

Thirdly, it was implemented an SQL Server database to account for the lack of connection to the SAP ERP system. The system's REST API module can connect directly with the SQL Server database and retrieve stored data.

Lastly, these three pipelines are handled by the system's REST API module and can be consumed by clients inside or outside Bosch's internal network. If the HTTP requests come from outside Bosch, the responses must pass through an additional layer of software, the Bosch API Manager. This software is responsible for exposing web APIs to the external network and implements additional security layers.

4.1 REST API

The REST API uses a WebApi template from Microsoft .NET 5.0 Framework. This template is a preconfigured C# project that implements a REST API with the MVC architecture. For this implementation, the module View of the MVC architecture is not necessary because the AR devices handle data visualization. For this reason, only the models and controllers were necessary. This template also brings a *Startup* and *Program* class to configure and initiate the web service.

The REST API also possesses *RestClient* and *SQLiteDataAcess* modules. These C# classes serve to consume information from different source types. Figure 4.2 shows the REST API's architecture.

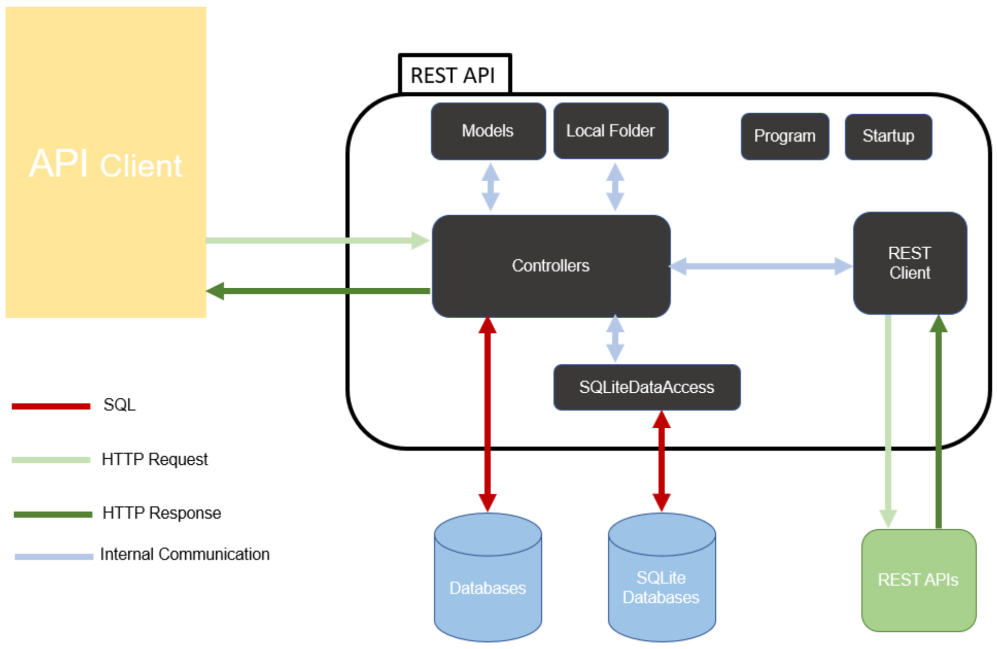
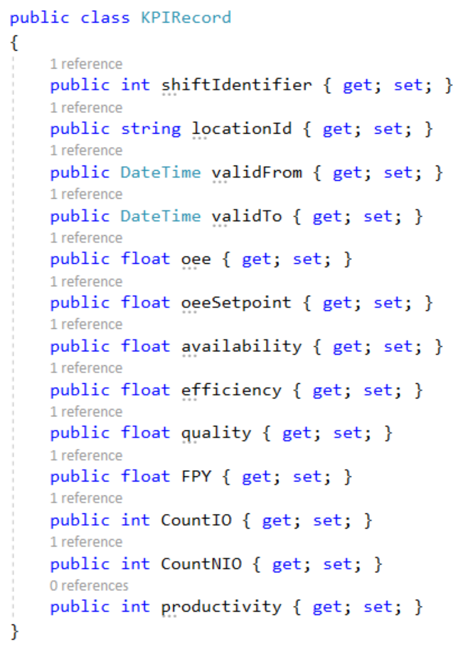


Figure 4.2: REST API Architecture

4.1.1 Models

Models represent tangible objects or notions, capable of having states and parameters that describe them. In this case, models are C# classes the program uses to create objects in the programming logic. An example of a model is the *KPIRecord*, shown in figure 4.3, which is necessary for use case 4.



```
public class KPIRecord
{
    1 reference
    public int shiftIdentifier { get; set; }
    1 reference
    public string locationId { get; set; }
    1 reference
    public DateTime validFrom { get; set; }
    1 reference
    public DateTime validTo { get; set; }
    1 reference
    public float oee { get; set; }
    1 reference
    public float oeeSetpoint { get; set; }
    1 reference
    public float availability { get; set; }
    1 reference
    public float efficiency { get; set; }
    1 reference
    public float quality { get; set; }
    1 reference
    public float FPY { get; set; }
    1 reference
    public int CountIO { get; set; }
    1 reference
    public int CountNIO { get; set; }
    0 references
    public int productivity { get; set; }
}
```

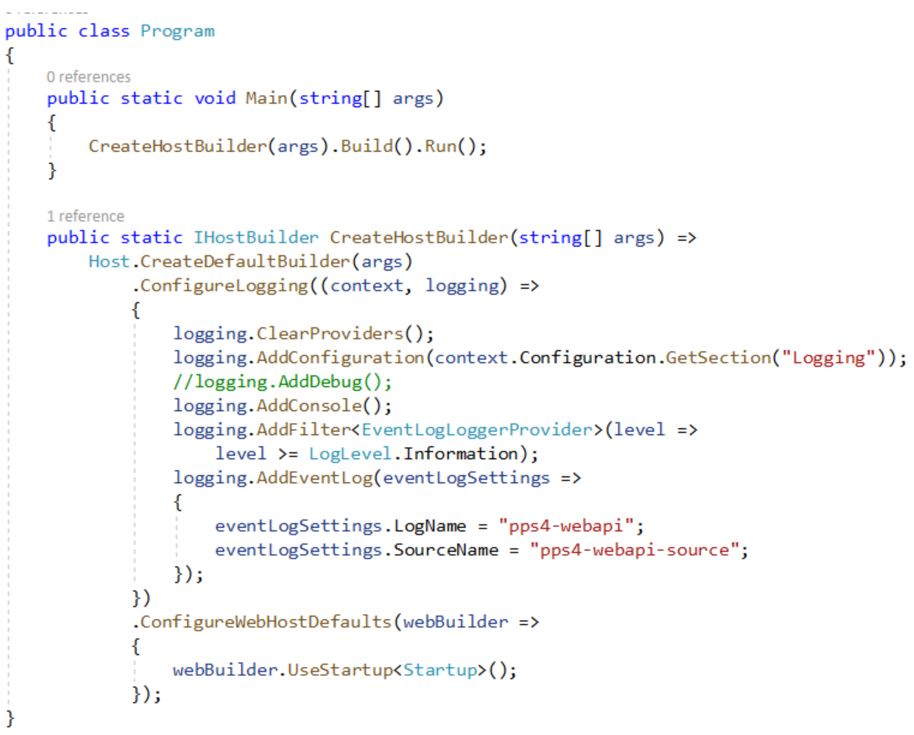
Figure 4.3: *KPIRecord* model

There are three types of models in this implementation. First, there are the models that mimic database tables. Their objective is to retrieve data from those tables so each model's property matches a table's column. The program creates an instance of the model and fills the properties with a single table row data. The second type is a model created to present information to the client, with properties that aim to fulfill the use case requirements in the information displayed. The last type is models used for internal processing that neither serve to retrieve nor show information.

4.1.2 Program

The *Program* class, in figure 4.4, belongs to the .NET Framework WebAPI template. This class's purpose is to initiate the REST API. *Program* has a method named *CreateHostBuilder()* that calls the function *CreateDefaultBuilder()*. This new function creates a new instance of *Microsoft.Extensions.Hosting.HostBuilder* class with settings pre-configured and loaded from *appsettings.json*. Chained to *CreateDefaultBuilder()*, it is executed the function *ConfigureWebHostDefaults* that executes the *Startup* class. This *Startup* class allows customizing the API further.

In this class, it is also necessary to configure the logging options. Loggings are essential because they are convenient for debugging the application once running in the background. It is possible to check the loggings in the Windows application Event Viewer under pps4-webapi.



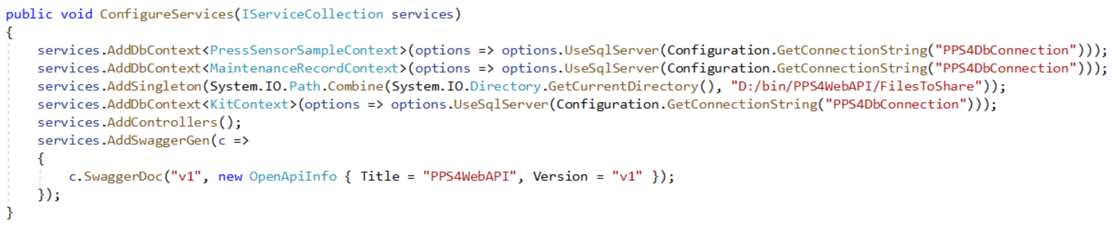
```
public class Program
{
    0 references
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    1 reference
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging((context, logging) =>
            {
                logging.ClearProviders();
                logging.AddConfiguration(context.Configuration.GetSection("Logging"));
                //logging.AddDebug();
                logging.AddConsole();
                logging.AddFilter<EventLogLoggerProvider>(level =>
                    level >= LogLevel.Information);
                logging.AddEventLog(eventLogSettings =>
                {
                    eventLogSettings.LogName = "pps4-webapi";
                    eventLogSettings.SourceName = "pps4-webapi-source";
                });
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Figure 4.4: REST API's *Program* class

4.1.3 Startup

The Startup class belongs to the .NET Framework WebAPI template, and it has the purpose of loading configurations in the application's launch. This class has two methods: *ConfigureServices()* and *Configure()*.

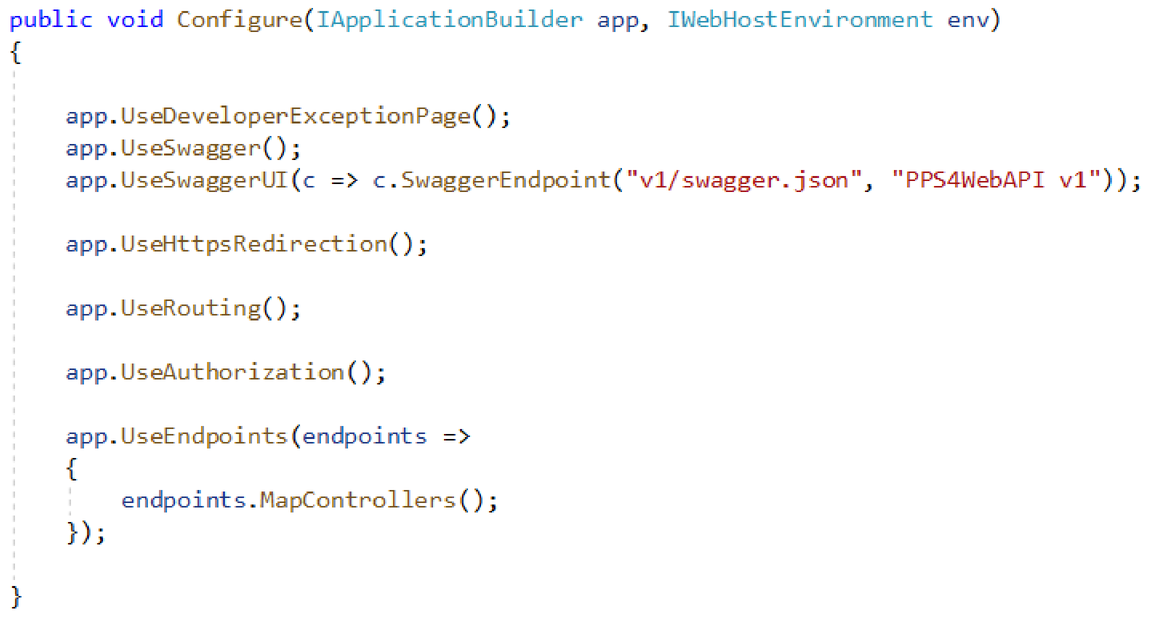
The *ConfigureServices()* method, figure 4.5, serves to add new services to the application. In this case, it imports *DbContexts* to open database connections and *Singletons* to add a path to a local folder. This method already possesses the *AddControllers()* function and *AddSwaggerGen()*, necessary to initiate the application. The first function maps the project, finds all classes labeled "Controllers", and initiates them as services. The second generates swagger-generated services, which will create documents containing the metadata of the HTTP messages.



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<PressSensorSampleContext>(options => options.UseSqlServer(Configuration.GetConnectionString("PPS4DbConnection")));
    services.AddDbContext<MaintenanceRecordContext>(options => options.UseSqlServer(Configuration.GetConnectionString("PPS4DbConnection")));
    services.AddSingleton(System.IO.Path.Combine(System.IO.Directory.GetCurrentDirectory(), "D:/bin/PPS4WebAPI/FilesToShare"));
    services.AddDbContext<KitContext>(options => options.UseSqlServer(Configuration.GetConnectionString("PPS4DbConnection")));
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "PPS4WebAPI", Version = "v1" });
    });
}
```

Figure 4.5: *ConfigureServices()* method

The *Configure()* method, figure 4.6, serves to add WebAPI-specific properties to the application. The first function, *UseDeveloperExeceptionPage()*, catches *System.Exception* instances and generates HTML error responses, and for this reason, it is completely optional. *UseSwagger()* and *UseSwaggerUI()* are necessary to use browser interface and are optional for REST client consumption. The first function creates a swagger.json file with all the relevant information about the system. The second function allows the system to create HTML files with a swagger template to show the application's controllers and endpoints. The fourth function, *UseHttpsRedirection()*, adds a middleware to redirect HTTP requests to HTTPS, which is optional. The *UseRouting()* function defines a point in the middleware pipeline to make routing decisions and associate the endpoints to their respective *HttpContexts*. The last object contains all the information about each HTTP request relative to the HTTP protocol. The *UseAuthoriztion()* allows the system to use authorization verifications present in the HTTP protocol. The last function, *UseEndpoints()*, uses the endpoints found in the controllers and creates a middleware to execute the endpoint relative to the current request.



```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{

    app.UseDeveloperExceptionPage();
    app.UseSwagger();
    app.UseSwaggerUI(c => c.SwaggerEndpoint("v1/swagger.json", "PPS4WebAPI v1"));

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });

}
```

Figure 4.6: *Configure()* method

4.1.4 SQLiteDataAccess

This class aims to access the SQLite database managed by the *KafkaConsumer* program. This class defines the connection string to the database and possesses two methods: *OpenConnection()* and *CloseConnection()*. These methods open and close the connection to the database in a safer form because they verify its state beforehand. This class has only the *con* property, which other classes use to execute SQL statements in the database.

4.1.5 REST Client

This class aims to create and send HTTP requests and retrieve the HTTP response. This class has four properties: *endPoint*, *httpMethod*, *user*, and *password*. If another project's class needs to get information from another application via HTTP, it creates a *RestClient* object and fills its properties. The *endPoint* should have the requested object's URI. The *httpMethod* defaults to the GET verb, but it can change to POST if needed. The user and password are optional, and they are only necessary if requesting applications that use authorization protocols. These credentials assume that the authorization method is *Basic Auth*.

4.1.6 Local Folder

The Local Folder is a directory located on the REST API server. This directory's purpose is to store the files necessary to use case 2 – Equipment Technical Data. The information about the machines and tools is very sensitive to Bosch, so it was impossible to connect REST API to the location of the machines and tools' stored data. The local folder compromises practicality because every file that needs to be exposed to the clients requires to be manually duplicated to the local folder. However, it is a viable solution facing the restrictions imposed.

4.1.7 Controllers

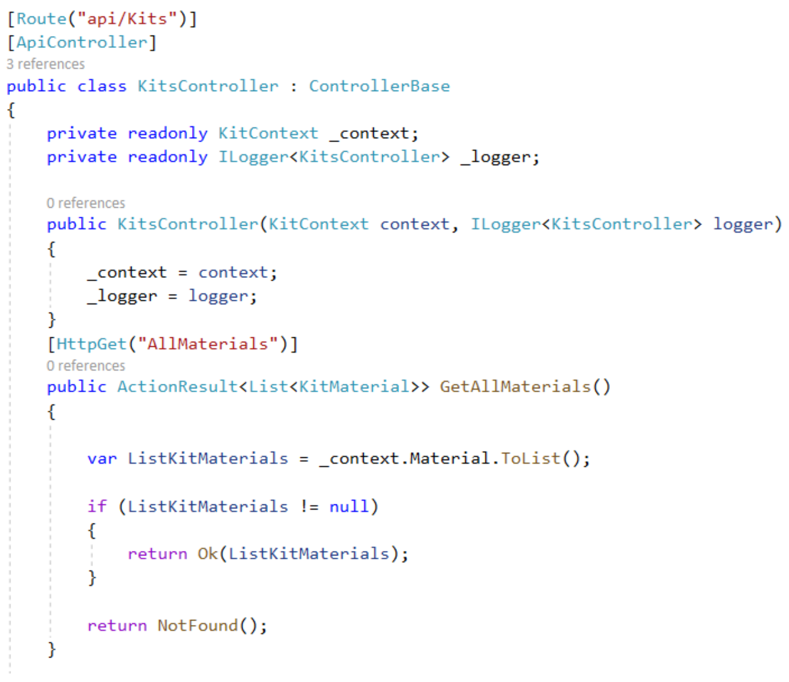
A controller is a procedural concept, and it is comparable to executable functions with parameters and returns values. In this case, a controller is a C# class decorated with an *ApiController* attribute that inherits methods from the *ControllerBase* class, as shown in figure 4.7. The primary route to access the controller's endpoints is *"api/"* plus the controller's name. A controller's endpoint is a public method decorated with an HTTP attribute inside the controller class. Figure 4.7 shows an endpoint decorated with a GET verb, adding to the main route *"/AllMaterials"*. To request this endpoint, the client adds *"api/kits/AllMaterials"* at the end of the URL. In this example, the code inside the method will execute and respond accordingly with a *200 OK* or *404 Not found* message.

Usually, the controllers manipulate instances of models. In this example, the variable *_context* has the configuration to output a list of instances of the *KitMaterial* model type. The *Ok()* function inherited from the *ControllerBase* class maps the *KitMaterial* properties and creates a JSON message with the same fields. In the case of List, this message is an array of JSON structures.

The REST API possesses six controllers and eighteen endpoints. Table 4.1 exposes the controllers and their respective endpoints with a brief description. The endpoints' complete URIs become *https://av-tef01-emea-com/PPS4/api/{endpoint}*, with {*endpoint*} being substituted with the first column of Table 4.1.

endpoint	description
Files/download/{id}	Uploads a file from the Local Folder to the client when given the file's id.
Files/hierarchy	Sends a JSON message containing the files in the Local Folder and their id.
Kits/Reserva/{nReserva}	Sends a JSON message containing data about the work order and the kits when given the {*nReserva*} which is the work order identification number.
Kits/Reserva/Last	Sends a JSON message containing data about the work order and kits of the last work order saved in the SQL server database.
Kits/AllMaterials	Sends a JSON message with the information of all the kits and materials stored in the SQL server database.
Kits/{idMaterial}	Sends a JSON message with the .information of kit or material when give the identification number.
kpi/cycle-times	Sends a JSON message with the information of the cycle times of final assembly lines' stations.
kpi/cycle-times/stations	Sends a JSON message informing which station's information is available.
kpi	Sends a JSON with the KPIs of a station for a time between two dates.
kpi/current-shift	Sends a JSON message with the KPIs of the current shift of a given station.
MaintenanceRecords	Sends a JSON message with all the maintenance records instances saved in the SQL server database.
MaintenanceRecords/ {id}	Sends a JSON message with the maintenance record saved in the SQL Server database with the given id.
MaintenanceRecords/ GetNewest/ {NewestX}	Sends a JSON message with the newest {*NewestX*} maintenance records instances saved in the SQL server database
MaintenanceRecords/ GetNewestFromDate/ {fromDateString}	Sends a JSON message with the maintenance records instances from {*fromDateString*} up to the current time.
MaintenanceRecords/ GetOldest/{OldestX}	Sends a JSON message with the oldest {*OldestX*} maintenance record saved in the SQL server database.
MaintenanceRecords/ GetDateRange/ {StartDateString}/ {EndDateString}	Sends a JSON message with the maintenance records instances from {*StartDateString*} up to {EndDateString}.
bottlenecks/actual	Sends a JSON message informing which station is the bottleneck in each line.
PressSensorSamples/ last-samples/{SamplesN}	Sends a JSON message with {*SamplesN*} instances of the press sensors' values stored in the SQLite database.

Table 4.1: REST API's endpoints descriptions



```
[Route("api/Kits")]
[ApiController]
3 references
public class KitsController : ControllerBase
{
    private readonly KitContext _context;
    private readonly ILogger<KitsController> _logger;

    0 references
    public KitsController(KitContext context, ILogger<KitsController> logger)
    {
        _context = context;
        _logger = logger;
    }
    [HttpGet("AllMaterials")]
    0 references
    public ActionResult<List<KitMaterial>> GetAllMaterials()
    {

        var ListKitMaterials = _context.Material.ToList();

        if (ListKitMaterials != null)
        {
            return Ok(ListKitMaterials);
        }

        return NotFound();
    }
```

Figure 4.7: Controller example

4.2 Kafka Broker Consumer

The Kafka Broker Consumer uses a *Worker* template from Microsoft .NET 5.0 Framework. The *Worker* template creates a generic service that allows the implementation of programming logic to run in the background of a machine. Figure 4.8 shows the Kafka Consumer architecture.

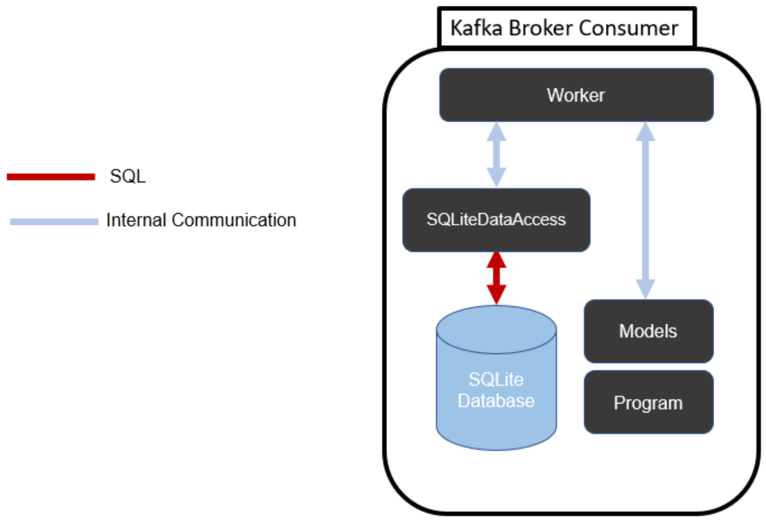


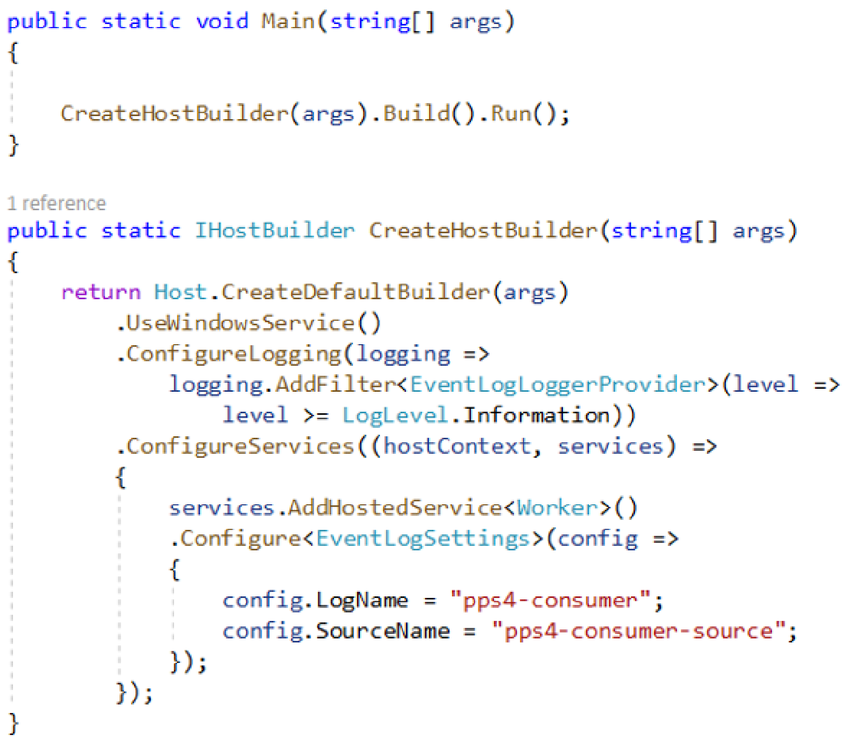
Figure 4.8: Kafka Consumer Architecture

4.2.1 Models

These models are similar to those described in subsection 4.1.1 in the REST API description. The models are C# classes with properties that try to emulate real-life objects or concepts. In this case, the models mimic the SQLite tables because the Kafka Consumer's only purpose is to store data coming from the Kafka Message Broker.

4.2.2 Program

The Program class, in figure 4.9, belongs to the .NET Framework *Worker* template. The purpose of this class is to initiate the *Worker* service. The *Main()* method calls the *CreateHostBuilder()* that creates an *IHostBuilder* instance with some pre-configured features. The addition of the *UseWindowsService()* function is necessary to use the *Worker* service application in the Windows Service software. This function belongs to the nugget package *Microsoft.Extensions.Hosting.WindowsServices*. Then, the configuration of the loggings with the *ConfigureLogging()* function and the service's configuration in *ConfigureService()*. The last function adds the *Worker* class to the *IHostBuilder* instance, which will execute when the builder initiates.

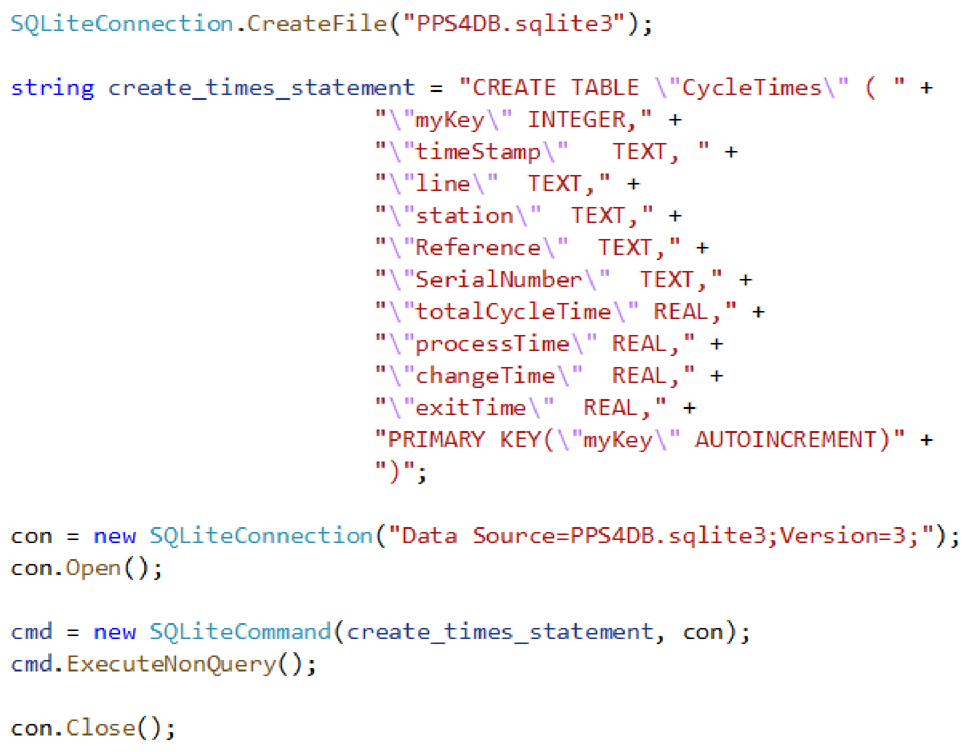


```
public static void Main(string[] args)
{

    CreateHostBuilder(args).Build().Run();
}

1 reference
public static IHostBuilder CreateHostBuilder(string[] args)
{
    return Host.CreateDefaultBuilder(args)
        .UseWindowsService()
        .ConfigureLogging(logging =>
            logging.AddFilter<EventLogLoggerProvider>(level =>
                level >= LogLevel.Information))
        .ConfigureServices((hostContext, services) =>
        {
            services.AddHostedService<Worker>()
            .Configure<EventLogSettings>(config =>
            {
                config.LogName = "pps4-consumer";
                config.SourceName = "pps4-consumer-source";
            });
        });
}
```

Figure 4.9: *KafkaConsumer*'s *Program* class

4.2.3 SQLiteDataAccess

The *SQLiteDataAccess* is a C# class that manages the connection to the SQLite database. The program checks if an SQLite database exists in the application's location. If there is no database, the program creates it and adds the necessary tables automatically. This class constructor executes every time the program creates a new *SQLiteDataAccess* instance. Figure 4.10 shows an example of the database and table creation.



```
SQLiteConnection.CreateFile("PPS4DB.sqlite3");

string create_times_statement = "CREATE TABLE \"CycleTimes\" ( " +
                                "\"myKey\" INTEGER," +
                                "\"timeStamp\"   TEXT,  " +
                                "\"line\"  TEXT," +
                                "\"station\"  TEXT," +
                                "\"Reference\"  TEXT," +
                                "\"SerialNumber\"  TEXT," +
                                "\"totalCycleTime\" REAL," +
                                "\"processTime\" REAL," +
                                "\"changeTime\"  REAL," +
                                "\"exitTime\"  REAL," +
                                "PRIMARY KEY(\"myKey\" AUTOINCREMENT)" +
                                ")";

con = new SQLiteConnection("Data Source=PPS4DB.sqlite3;Version=3;");
con.Open();

cmd = new SQLiteCommand(create_times_statement, con);
cmd.ExecuteNonQuery();

con.Close();
```

Figure 4.10: SQLite database and table creation

This class has two methods: *OpenConnection()* and *CloseConnection()*. The methods' purpose is to open and close the connection to the database safely, as they check the connection's state before applying any action.

4.2.4 Worker

The *Worker* class contains the code running in the background as a service, and figure 4.11 shows its behavior with an activity diagram.

In the application's initiation, the class loads the initial variables in its constructor. The values of the variables are hardcoded because they do not need to change. These properties are *HOST*, *TOPIC_8220*, *TOPIC_7*, *TOPIC_10* and *CONSUMER_GROUP_ID*. Then, the program configures the *Worker* using the properties and subscribes to the topics *TOPIC_8220*, *TOPIC_7*, and *TOPIC_10*, which correspond to *Line8220*, *Line7*, and *Line10*. After the subscription, the program enters a while loop and stays listening to incoming messages. In this listening state, the program checks if there are new messages on any topic.

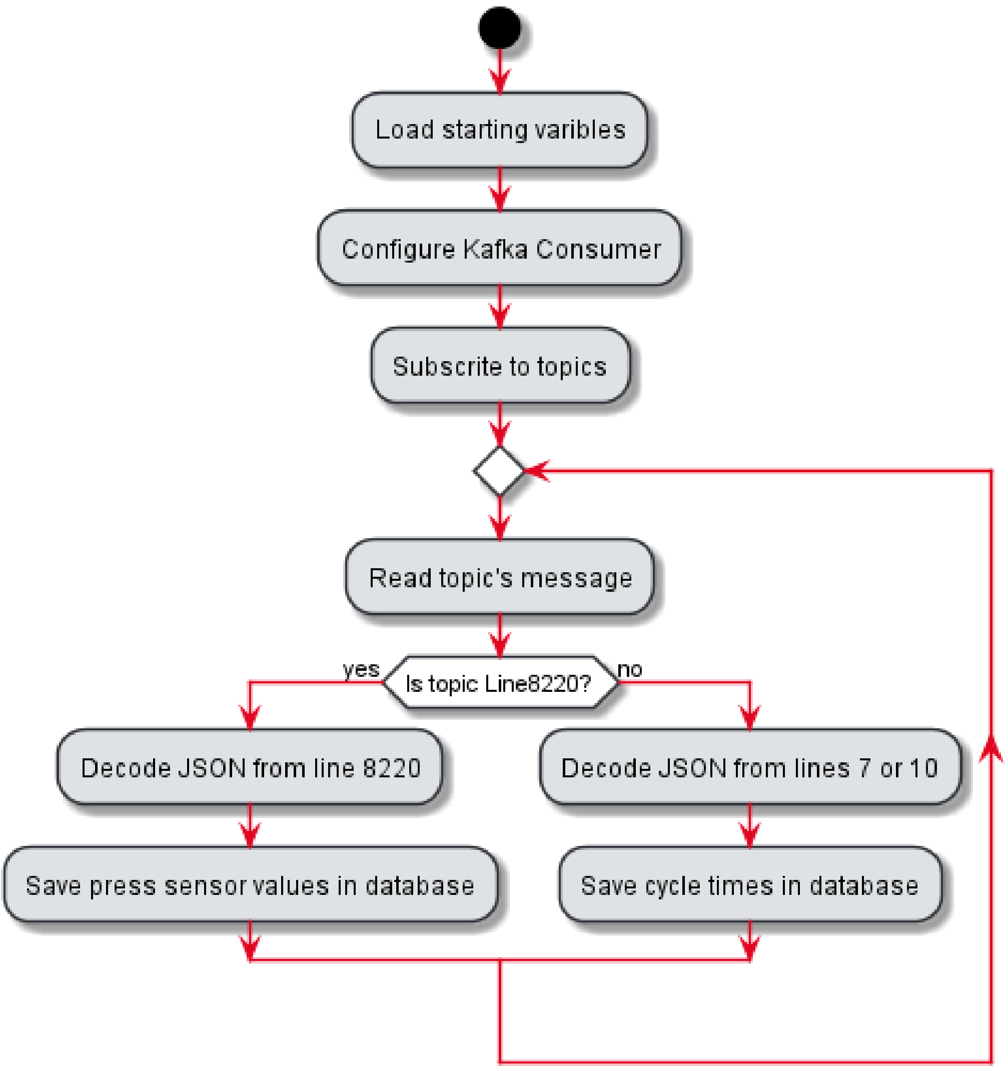

Load starting varibles
Configure Kafka Consumer
Subscrite to topics
Read topic's message
yes
Is topic Line8220?
no
Decode JSON from line 8220
Decode JSON from lines 7 or 10
Save press sensor values in database
Save cycle times in database

Figure 4.11: *Worker* logic activity diagram

If there is a new message, and it comes from topic *Line8220*, it decodes the JSON and searches for the press sensor values, mapping them to a *PressSensorSample* instance. Then, it saves the data in the *PressSensorValues* table. While the program saves the new information, it deletes old information, never to have more than three hundred entries stored. The database keeps only small storage because this implementation's purpose is to be real-time, and three hundred entries translate to roughly five minutes of data.

If the new message does not come from topic *Line8220*, it must come from *Line7* or *Line10*. These two topics output JSON messages with similar formats, so the same functions are used to decode and store the data. The *Worker* class stores the information in the *CycleTimes* table. The requirement for this implementation is to display only the cycle time data relative to the current working shift. For that reason, the program deletes any entries relative to previous shifts.

4.3 Nexeed MES Communication Pipeline

In Bosch, the Nexeed MES possesses some APIs to retrieve data from the Oracle Server. The MES database has stored procedures used by the other MES servers and third-party software to access data. This implementation uses the PDA/MDA's API, specifically the endpoints that call the stored procedures *PC01* and *OE01*. These stored procedures return part counts and KPI, which are necessary to use case 3 – Final Assembly Line. Figure 4.12 shows this communication pipeline.

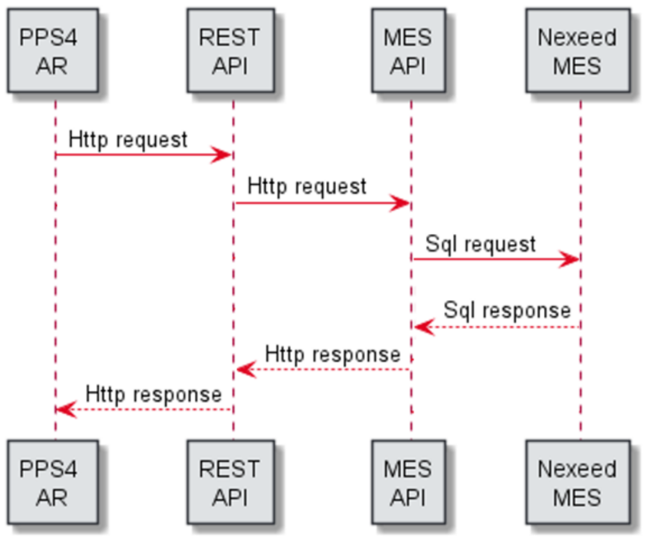


Figure 4.12: Nexeed MES Communication Pipeline

This communication occurs in the *KPIsController* class in the REST API application. The controller creates a *RestClient* instance and uses it to send HTTP requests containing the two endpoints mentioned before. Then, the controller filters the information received and maps the relevant data to a *KPIRecord* instance.

Consuming an API instead of connecting directly to a database is safer. In production, many new applications need to connect to the production database. The idea is to have a protective middleware that does not allow third-party software to connect directly to applications that the production environment needs to function.

4.4 Kafka Message Broker Communication Pipeline

The web service acquires information from the shop floor using the Kafka Message Broker. Figure 4.14 shows the communication stream between the SQLite Database and the AR devices and figure 4.13 shows the architecture of the comunication that provides information to the SQLite database.

DirectDataLink (DLL) is an interface with the primary function of establishing communication between the shop floor's machines and Nexeed MES. This middleware is a Windows Services that receives data in *OpCon XML* protocol, *.dat* files, and *OpcUA* and redirects it to other applications. This system also sends the information to an Apache

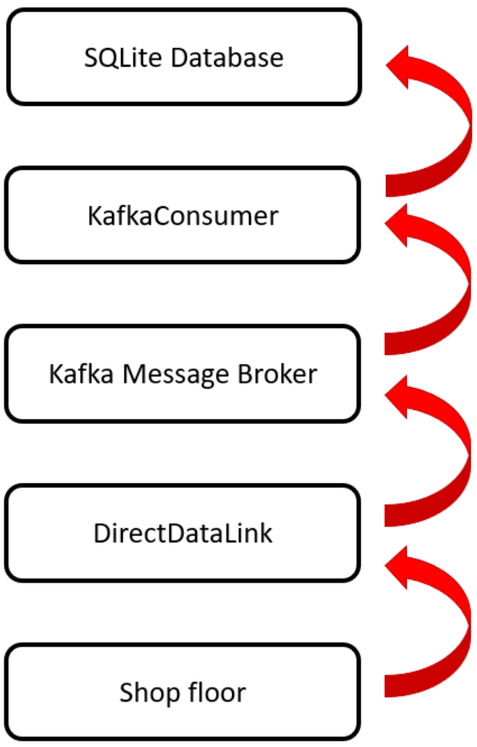


Figure 4.13: Kafka Message Broker Communication Pipeline

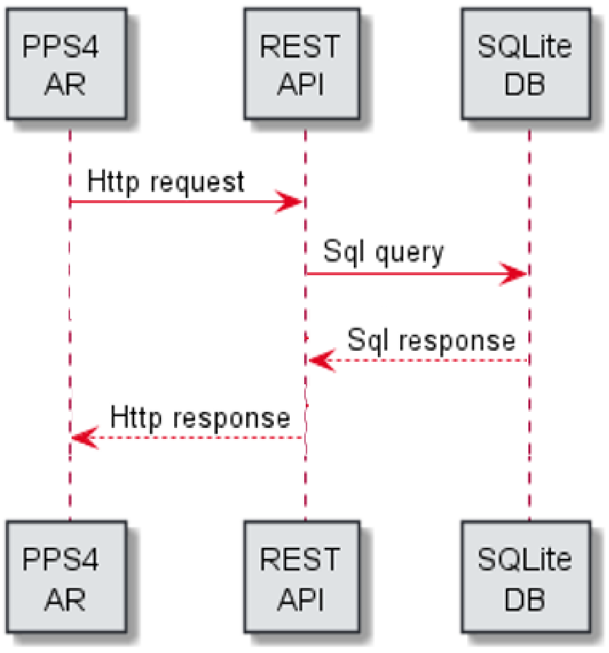


Figure 4.14: Kafka Message Broker Communication Pipeline

Kafka Message Broker as part of another communication stream. The implemented system uses this data stream to obtain information about the shopfloor. The Kafka Message Broker cannot publish messages to topics in XML format, so it maps the information to JSON messages with similar fields. These topics are available to applications to consume, and almost every line has one topic. The web service's *KafkaConsumer* is one of the applications that consume data from the Kafka Message Broker. Subsection 4.2.4 explains that the program reads relevant topics and stores the messages in a local SQLite database. This database is always accessible to the REST API module. Two controllers use the data stored in the SQLite database: *KPIsController* and *PressSensorSamplesController*. Both controllers have a similar behavior when it is requested information from the Kafka Consumer:

1. The program creates a *SQLiteDataAccess* instance, described in subsection 4.2.3, and opens a connection to the database.
2. It reads information from the *CycleTimes* or the *PressSensorValues* table, respectively.
3. It maps the information to a *KPIRecord* or *PressSensorSample* instance and sends an HTTP response to the client.

4.5 SQL Server Communication Pipeline

This communication pipeline consists of an SQL Server database that the REST API module can access. This implementation serves to emulate data from SAP ERP because, due to some restrictions, it was impossible to establish a connection between the web service to the SAP ERP database in time. The solution was to create a database with tables capable of storing the information necessary for use case 2 – Equipment Technical Data and use case 3 – Assisted Production. The data stored in the database is information about the assembling kits and the Haulick & Roos RVD200T press maintenance records. Three separate tables store the information about the kits: *OrdemTrabalho*, *Material*, and *RelacaoMateriais*. Figure 4.15 show the tables' relation diagram. The *RelacaoMateriais* does not have a direct relation between the other two tables, but it has an indirect relation with the values stored in the *BOM* field in the table *Material*.

The separation of the kits' storage was due to its complexity. A kit, also considered a material, possesses other materials in its contents. The kit's bill of materials (BOM) lists the kit's materials. A kit can have another kit as its material, creating a nesting of kits that can have indefinite layers. For this reason, all available materials and kits storage are in the Material table. The relation between kits and their materials is stored in the *RelacaoMaterials* table. This table mainly informs about the materials' quantities inside the kit. Finally, the *OrdemTrabalho* table stores data relative to the workers' work orders during the shifts.

The controller that uses this pipeline is the *KitsController*. The endpoint *api/Kits/Reserva/{nReserva}* returns information about a kit having as input the identification number of a work order. The executed code's logic is shown in figure 4.16.

Firstly, the function reads the *OrdemTrabalho* table with the work order identification number and gets the kit's identification number and the number of kits that must be processed. Then, the function searches for information about the kit in the Material

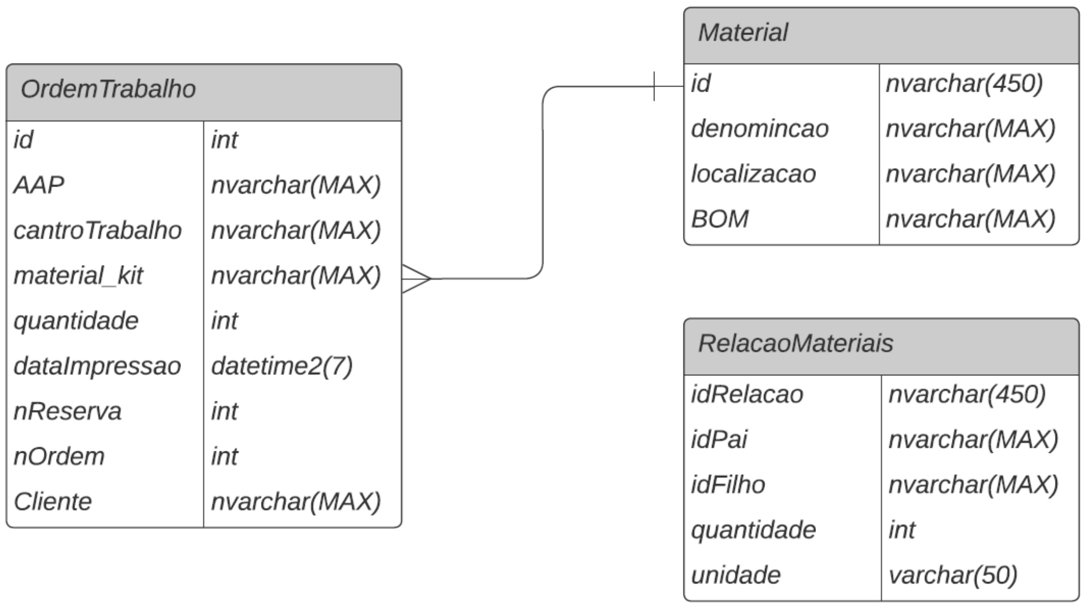


Figure 4.15: Tables' relation diagram

table and gets the kit's *BOM*. After that, using the kit's *BOM* searches for information about the kit's materials. This step checks if one of the new materials is also a kit. The next step is reading the quantities of the materials in the *RelacaoMaterials* table. Then, if a new kit exits in the kit's materials, the program repeats the last three steps for the new kit. When no more new kits exit, the function sends the information in an HTTP response.

4.6 Bosch External Network Communication

In the developing stages of Augmented Reality software, Bosch works with other entities that do not have access to Bosch's internal network. For this reason, it is necessary to expose the REST API's functionalities to outside networks. Bosch API Manager is a program implemented in Bosch's network that exposes Web APIs to different networks. There are three network layers in the company, which will be mentioned by *Layer one (L1)*, *Layer two (L2)* and *Layer three (L3)*.

The web service's server belongs to the *L3* network, and only the *L1* has a connection to external networks. For this reason, the REST API needs to be mapped by the three Bosch API Managers before reaching an external client. Figure 4.17 demonstrates this communication sequence implemented.

The first step was mapping the REST API to *L3*'s Bosch API Manager (BAM). The BAM mimics a REST API and its endpoints. After that, the application can redirect HTTP requests to the selected backend API and redirects the HTTP response to the next layer. The selected backend API is web service's REST API in the BCN case. In the API's frontend occurs the implementation of HTTP security protocols. The security protocol is the *API key*, which is easy to use but less safe. Whoever, this protocol will

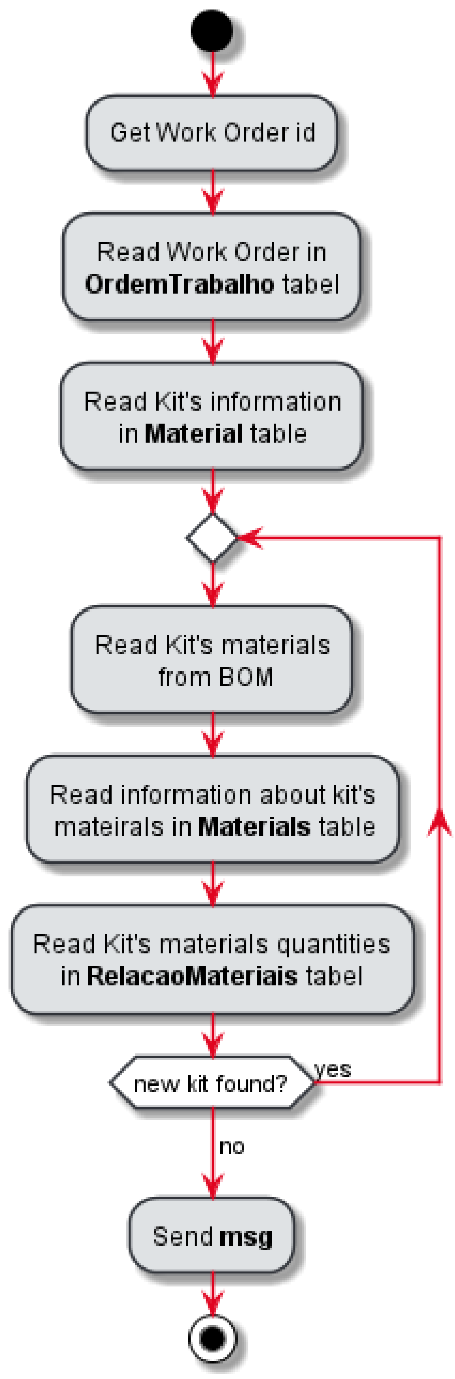

Get Work Order id
Read Work Order in **OrdemTrabalho** tabel
Read Kit's information in **Material** table
Read Kit's materials from BOM
Read information about kit's mateirals in **Materials** table
Read Kit's materials quantities in **RelacaoMateriais** tabel
new kit found?
yes
no
Send **msg**

Figure 4.16: *KitsController* endpoint's function logic

only be used in internal Bosch communications.

The second step was mapping the *L3*'s BAM to the *L2*'s BAM. The process is similar to the first step, and the *L2*'s BAM maps the endpoints from the previous API and then establishes a security protocol. The security protocol is the *API key* because the communication is in Bosch's internal network.

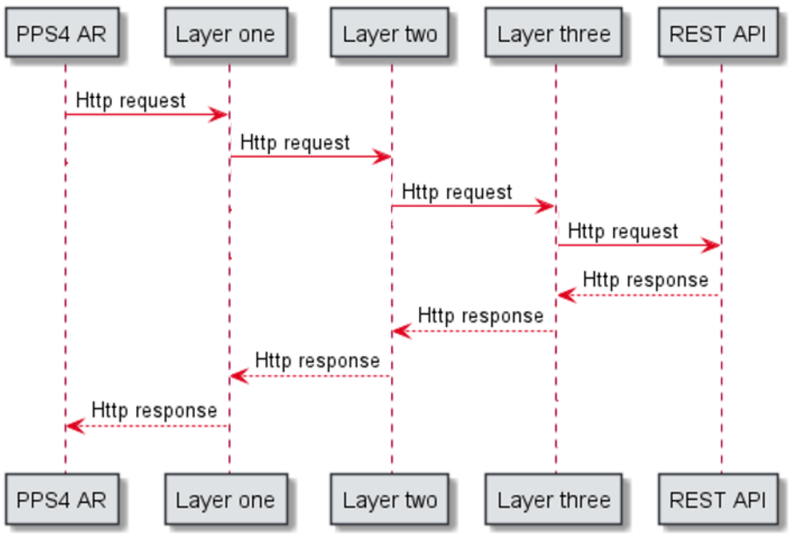


Figure 4.17: Bosch's API communication sequence

The final step was mapping the *L2*'s BAM to the *L1*'s BAM. The backend maps the endpoints from *L2*'s API. In the frontend, the security protocol implemented is HTTP *basic authentication*. This security protocol is safer than the previously implemented, and it is the recommendation for outside networks.

4.7 Web Service Production Implementation

The Web Service's location is the av-tef01 server, a server used by the AvP/MFD team. The system is composed of two modules which are essentially background services. For this reason, other applications must manage the services.

The first application is the Window's Internet Information Services (IIS) Manager. This software aims to handle services that communicate via HTTP in a server-client architecture. This application configures the SSL certification, hostname, and site name. The IIS Manager stays listening to HTTP requests with the services URL and redirects them to the REST API. After that, the software also handles the HTTP responses provided by the REST API, adding the necessary metadata.

The second application is the Window's Services application. This software is more generic than the previously presented and allows the execution of any program with a background service architecture. The Services application manages the Kafka Consumer module.

The two Web Service's modules are programmed to log any relative information about behaviors and errors of the applications. Logging is crucial to background services, as they usually do not have any other output. If a program does not log any information about its current state, it can be challenging to understand problems that might happen in the future. The logs provided by the programs are present in the Window Application Event Viewer.

Intentionally blank page.

Chapter 5

Test Solution

This chapter aims to explain the solution's validation. Testing the solution helps to justify the methods used or to understand what could be improved. The tests focused on the REST API because it is the project's main module and the component that interacts with the user.

There are many ways to test web services by measuring their security, performance, reusability, efficiency, reliability, interoperability, and maintainability [38] [39]. However, the guiding requirement throughout the project was the real-time data exposition, and for this reason, the performance of the system was the focus of the testing section.

The performance of a system is related to the quality and time needed to complete a request. There were two methods for this testing: endpoint and load testing. These tests were performed by simulating web consumers inside the internal Bosch network, which the requests were sent by the computer, passed through routers and reached the web service, as is shown in figure 5.1. For safety reasons, the information about the internal Bosch network and the specifications of the devices cannot be provided.

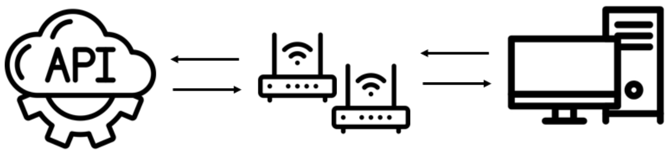


Figure 5.1: Communication setup for the test

For each test, it will be presented the average time, in milliseconds, between request and response and a confidence interval CI of expected times with a confidence level of 95%, t_i a time sample, μ the mean of the measured times, and σ the standard deviation. These values will be shown for each endpoint of the REST API. Table 5.1 presents the number associated with each endpoint to save space in the following tables. The values in the next sections were obtained using equations 5.1 and 5.2.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(t_i - \mu)^2}{}} \tag{5.1}$$

$$CI = 1,96.\frac{\sigma}{\sqrt{N}} \tag{5.2}$$

Number	Endpoint
1	Files/download/id
2	Files/hierarchy
3	Kits/Reserva/nReserva
4	Kits/Reserva/Last
5	Kits/AllMaterials
6	Kits/idMaterials
7	kpi/cycle-times
8	kpi/cycle-times/stations
9	kpi
10	kpi/current-shift
11	MaintenanceRecords
12	MaintenanceRecords/id
13	MaintenanceRecords/GetNewest/NewestX
14	MaintenanceRecords/GetNewestFromDatefromDateString
15	MaintenanceRecords/GetOldest/OldestX
16	MaintenanceRecords/GetDateRange/StartDateString/EndDateString
17	bottlenecks/actual
18	PressSensorSamples/last-samples/SamplesN

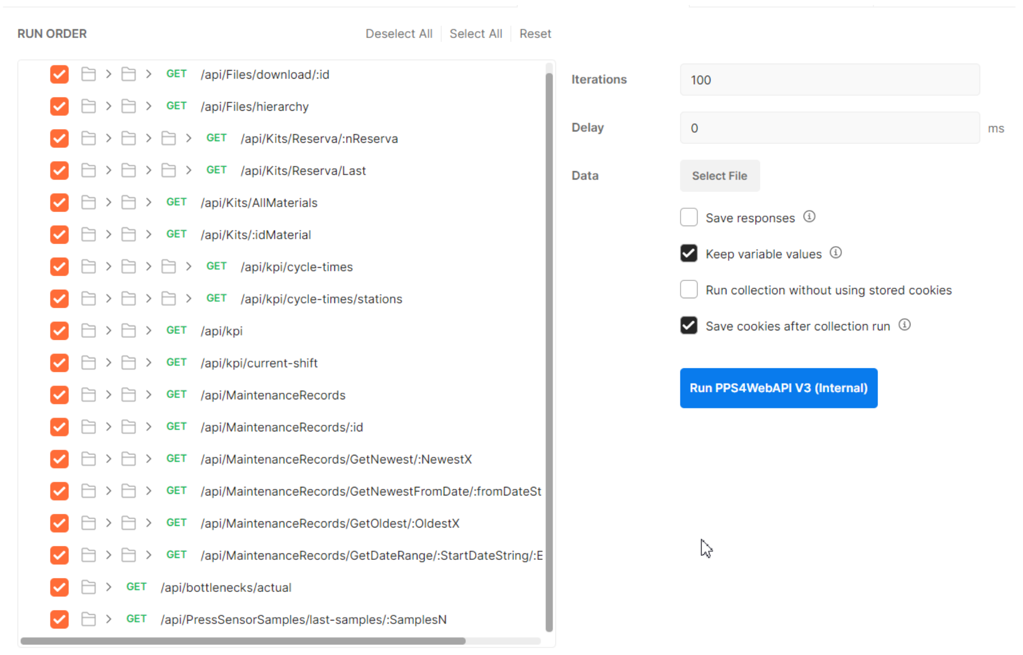
Table 5.1: Endpoint numerical association

5.1 Custom Testing

Endpoint testing is the custom testing scenario to measure ideal response times to compare to the load testing. Each endpoint is requested individually in this test, and the web service deals with one request at a time. The program Postman allowed to carry this testing type. This program is an API platform to use and consume APIs. This application allows the creation of a routine, figure 5.2, that calls the endpoints sequentially, saves the result and times, and then allows the export of it in JSON format.

After obtaining the result, a C# program read the JSON and calculated the values mentioned previously. Table 5.2 shows the calculated values.

Table 5.2 shows that the Postman requested one hundred times each endpoint. Every endpoint possessed a response time under 40 ms on average and under 50 ms maximum time with a confidence level of 95%, except for endpoint 1. The reason for that is the message size. This endpoint deals with files that tend to be more prominent in size than the JSON messages from the other endpoints. However, file downloading was not requested to be real-time, and downloading a file under 3 seconds is still practical in a production context.



RUN ORDER — Deselect All | Select All | Reset

- [✓] GET /api/Files/download/:id
- [✓] GET /api/Files/hierarchy
- [✓] GET /api/Kits/Reserva/:nReserva
- [✓] GET /api/Kits/Reserva/Last
- [✓] GET /api/Kits/AllMaterials
- [✓] GET /api/Kits/:idMaterial
- [✓] GET /api/kpi/cycle-times
- [✓] GET /api/kpi/cycle-times/stations
- [✓] GET /api/kpi
- [✓] GET /api/kpi/current-shift
- [✓] GET /api/MaintenanceRecords
- [✓] GET /api/MaintenanceRecords/:id
- [✓] GET /api/MaintenanceRecords/GetNewest/:NewestX
- [✓] GET /api/MaintenanceRecords/GetNewestFromDate/:fromDateSt
- [✓] GET /api/MaintenanceRecords/GetOldest/:OldestX
- [✓] GET /api/MaintenanceRecords/GetDateRange/:StartDateString/:E
- [✓] GET /api/bottlenecks/actual
- [✓] GET /api/PressSensorSamples/last-samples/:SamplesN

Iterations: 100

Delay: 0 ms

Data: Select File

- [ ] Save responses
- [✓] Keep variable values
- [ ] Run collection without using stored cookies
- [✓] Save cookies after collection run

Figure 5.2: Postman routine for costum test

endpoint	μ (ms)	CI (ms)	Maximum time (ms)	N
1	2269	392	2661	100
2	12	1,6	13,6	100
3	14	1,4	15,4	100
4	14	1,1	15,1	100
5	11	1,1	12,1	100
6	11	1,6	12,6	100
7	12	1,3	13,3	100
8	13	1,7	14,7	100
9	24	1,4	25,4	100
10	40	1,7	41,7	100
11	15	0,8	15,8	100
12	12	0,8	12,8	100
13	14	1,6	15,6	100
14	10	0,6	10,6	100
15	14	1,6	15,6	100
16	12	0.9	12.9	100
17	9	0,7	9,7	100
18	27	6,1	33,1	100

Table 5.2: Endpoints test results

5.2 Load Testing

Load testing aims to measure the performance of a web service with the same demand as the production state. This simulation is under the assumption that the three use cases are consuming the REST API simultaneously. For use case 1 – Equipment Technical Data, usually only one maintenance technician will use an AR device at a time. For use case 2 – Assisted Production, all workers of section S854 would use the AR device, usually four people. Lastly, only one line manager in lines 7 or 10 will use an AR device for use case 3 – Final Assembly Line. The expectation is to have six AR devices consuming the REST API simultaneously. Twelve clients were simulated in the load test to ensure the system's functionality, double the number of devices expected in the working environment.

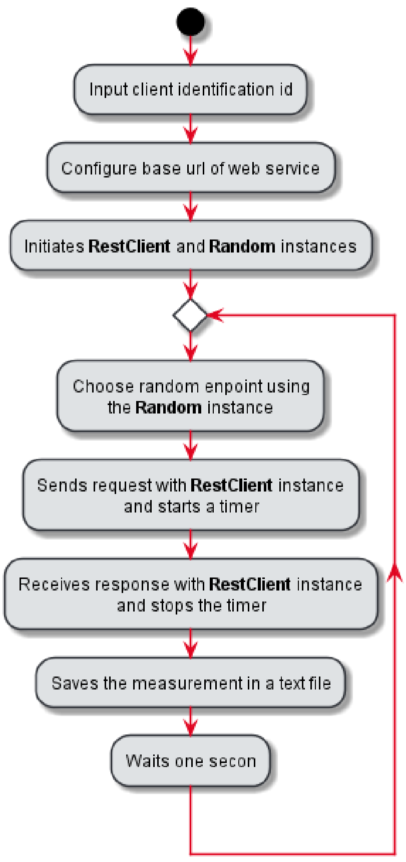

Input client identification id
Configure base url of web service
Initiates **RestClient** and **Random** instances
Choose random enpoint using the **Random** instance
Sends request with **RestClient** instance and starts a timer
Receives response with **RestClient** instance and stops the timer
Saves the measurement in a text file
Waits one secon

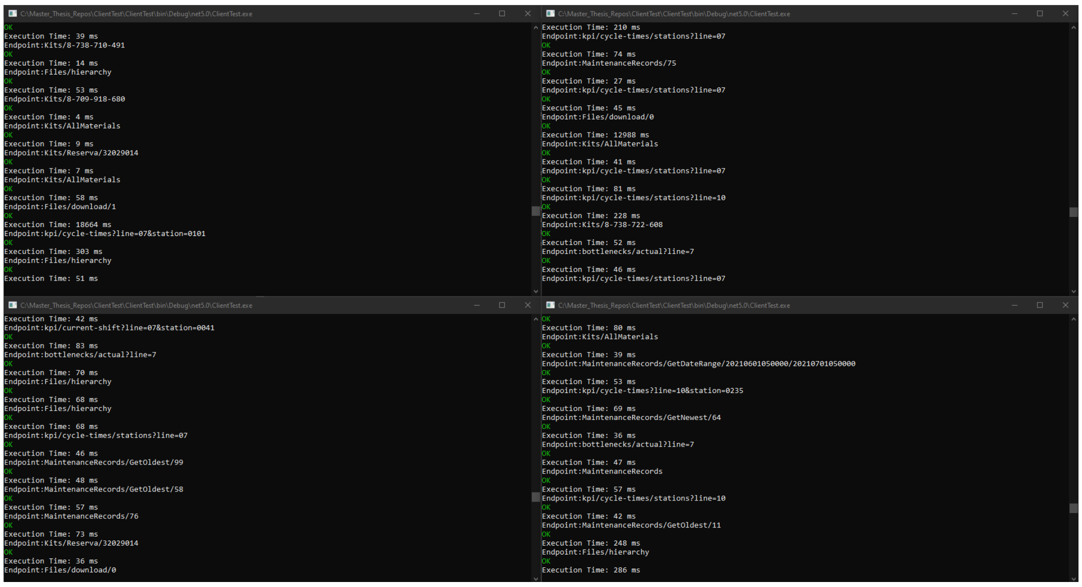
Figure 5.3: Client's activity diagram for load testing

An AR device simulation used a C# program. Figure 5.3 shows the program's activity diagram. This application configuration was to request a random endpoint and measure the time necessary to get a response. After receiving a response, the program waited one second before requesting another random endpoint.

The testing involved launching twelve different instances of the same client program explained previously. The testing took one hour and produced the following results in Table 5.3. This table exposes the client's results that obtained the maximum waiting time for each endpoint.

endpoint	μ (ms)	CI (ms)	Maximum	N
1	8167	1543	9710	140
2	57	11	68	149
3	54	9,8	53,8	145
4	64	13,3	77,3	155
5	54	11,5	65,1	118
6	52	9,9	61,9	123
7	57	8,8	65,8	118
8	56	9,4	65,4	134
9	109	20,7	129,7	95
10	96	11,5	107,5	100
11	94	25,8	119,8	126
12	56	8,7	64,7	126
13	83	20,9	103,9	136
14	56	10,7	66,7	128
15	89	26,3	115,3	122
16	60	15,4	75,4	145
17	50	10,3	60,3	136
18	145	35,6	180,6	141

Table 5.3: Load test results



```
C:\Master_Thesis_Repos\ClientTest\ClientTest\bin\Debug\net5.0\ClientTest.exe
OK
Execution Time: 39 ms
Endpoint:Kits/8-738-710-491
OK
Execution Time: 14 ms
Endpoint:Files/hierarchy
OK
Execution Time: 53 ms
Endpoint:Kits/8-709-918-680
OK
Execution Time: 4 ms
Endpoint:Kits/AllMaterials
OK
Execution Time: 9 ms
Endpoint:Kits/Reserva/32029014
OK
Execution Time: 7 ms
Endpoint:Kits/AllMaterials
OK
Execution Time: 58 ms
Endpoint:Files/download/1
OK
Execution Time: 18664 ms
Endpoint:kpi/cycle-times?line=07&station=0101
OK
Execution Time: 303 ms
Endpoint:Files/hierarchy
OK
Execution Time: 51 ms
```

```
C:\Master_Thesis_Repos\ClientTest\ClientTest\bin\Debug\net5.0\ClientTest.exe
Execution Time: 210 ms
Endpoint:kpi/cycle-times/stations?line=07
OK
Execution Time: 74 ms
Endpoint:MaintenanceRecords/75
OK
Execution Time: 27 ms
Endpoint:kpi/cycle-times/stations?line=07
OK
Execution Time: 45 ms
Endpoint:Files/download/0
OK
Execution Time: 12988 ms
Endpoint:Kits/AllMaterials
OK
Execution Time: 41 ms
Endpoint:kpi/cycle-times/stations?line=07
OK
Execution Time: 81 ms
Endpoint:kpi/cycle-times/stations?line=10
OK
Execution Time: 228 ms
Endpoint:Kits/8-738-722-608
OK
Execution Time: 52 ms
Endpoint:bottlenecks/actual?line=7
OK
Execution Time: 46 ms
Endpoint:kpi/cycle-times/stations?line=07
```

```
C:\Master_Thesis_Repos\ClientTest\ClientTest\bin\Debug\net5.0\ClientTest.exe
Execution Time: 42 ms
Endpoint:kpi/current-shift?line=07&station=0041
OK
Execution Time: 83 ms
Endpoint:bottlenecks/actual?line=7
OK
Execution Time: 70 ms
Endpoint:Files/hierarchy
OK
Execution Time: 68 ms
Endpoint:Files/hierarchy
OK
Execution Time: 68 ms
Endpoint:kpi/cycle-times/stations?line=07
OK
Execution Time: 46 ms
Endpoint:MaintenanceRecords/GetOldest/99
OK
Execution Time: 48 ms
Endpoint:MaintenanceRecords/GetOldest/58
OK
Execution Time: 57 ms
Endpoint:MaintenanceRecords/76
OK
Execution Time: 73 ms
Endpoint:Kits/Reserva/32029014
OK
Execution Time: 36 ms
Endpoint:Files/download/0
```

```
C:\Master_Thesis_Repos\ClientTest\ClientTest\bin\Debug\net5.0\ClientTest.exe
OK
Execution Time: 80 ms
Endpoint:Kits/AllMaterials
OK
Execution Time: 39 ms
Endpoint:MaintenanceRecords/GetDateRange/20210601050000/20210701050000
OK
Execution Time: 53 ms
Endpoint:kpi/cycle-times?line=10&station=0235
OK
Execution Time: 69 ms
Endpoint:MaintenanceRecords/GetNewest/64
OK
Execution Time: 36 ms
Endpoint:bottlenecks/actual?line=7
OK
Execution Time: 47 ms
Endpoint:MaintenanceRecords
OK
Execution Time: 57 ms
Endpoint:kpi/cycle-times/stations?line=10
OK
Execution Time: 42 ms
Endpoint:MaintenanceRecords/GetOldest/11
OK
Execution Time: 248 ms
Endpoint:Files/hierarchy
OK
Execution Time: 286 ms
```

Figure 5.4: Clients receiving messages in load testing

5.3 Result Analysis

REST API is ideal for single-user real-time data transfer, as all real-time data had a delay under 50 ms. For the load testing, the delay values increased as expected. This testing had a very conservative approach, doubling the amount of expected users and simulating that each user would make a request each second, which is an higher frequency of request than normal. Despite that, the maximum delay expected is under 200ms for every endpoint besides the download file endpoint. For this implementation, and because it is a wireless connection, guaranteeing a 200ms maximum delay is enough to fulfil the real-time requirement.

The endpoint 1 needs to be analyzed separately. As the tests showed, the maximum time delay in downloading a document is 10 seconds. However, it depends on the size of the file. The file with the most considerable size used in the testing had 50 megabytes, but the expectation is that files with sizes in the order of gigabytes would take much longer to send. However, this endpoint does not require real-time data.

In conclusion, the test validated the implemented architecture for the endpoints that retrieved JSON messages with low data sizes and showed no significant difference between consuming information from an SQL server database, Kafka Messaging Broker or an SQLite database. However, this system may not be ideal for large file transfers.

Chapter 6

Conclusion

6.1 Conclusions

Throughout the development of this project, it was possible to experiment and understand different software and technologies to achieve the goal of real-time communication in an industrial context. It was not an easy task to create a system that uniformizes data transfer from all over the plant and delivers it steadily and reliably; however, some technologies and concepts facilitated the implementation.

Firstly, this project exposes the usefulness of Frameworks. In this system, both modules of the REST API and the Kafka Consumer used templates from .NET Framework, and this project would not have been completed in such a short time if it was not for this Framework. These templates helped implement the applications' necessary base behaviours, so the project's focus remained on the use case requirements.

Secondly, using SQLite database as internal storage for industrial software is a method that does not appear very often in documentation; however, it revealed itself to be a good solution for small data storage.

Thirdly, this project shows a different way of using Web APIs. Typically, a Web API serves as an interface of another web server application, mapping its functions and data sets. However, in this implementation, the REST API fulfils the web server's role and maps the information throughout Bosch's software with favourable results.

In the case of the use cases, not every condition was fulfilled. The most crucial requirement that was not fulfilled was the SAP ERP connection. This connection enabled retrieval of information for use case 2 – Assisted Production, which corresponded to the kits for S854, and use case 1 - Equipment Technical Data, which corresponded to the maintenance records of the Haulick & Roos RVD200T. The connection was not established because of the restriction of direct connection to the software, and because of the time the project occurred, there was no safe way to connect to SAP ERP indirectly. However, a SQL server database with tables containing fake data was implemented, which serves as a proof-of-concept to the specific endpoints that retrieved the SAP ERP data. On the other side, it was found a way to retrieve the information from Nexeed MES and the Shopfloor to the communication system to fulfil the rest of the requirements. These requirements belonged to use case 2, which needed real-time sensor data of the stamping press, and case 3 - Final Assembly Line - Real-Time Monitoring, which needed the KPIs and cycle times of the lines 7 and 10.

The implementation aspect that needs to be handled with care is the *"Files/download*

/{id}" endpoint. This endpoint is very dependent on the file size when the message size of the other endpoints is fixed. The scientific documentation did not recommend using REST API for file transferring. However, in the testing section of the project, it was concluded that the time delay for the file downloading was acceptable for the context. Nevertheless, the uncertainty of future file sizes can generate difficulties.

In summary, this project was developed to respond to the Augmanity use cases requests, which were to retrieve data from different points of the company in real-time. Most of the requirements were fulfilled, and those not fully implemented have endpoints for future implementation.

6.2 Future Work

Systems like the one developed in this project are never finished because they can constantly be improved. The first improvement step is implementing the SAP ERP connection when it becomes available in Bosch's environment. Although this implementation is halfway done, some challenges should arise regarding the data consumption because some assumptions could not be accurate when creating the proof-of-concept endpoints, like the assumption of the models structure of the stored data. The next step would be finishing the bottleneck controller that depends on projects regarding the PPS2 - Big Data and Predictive Analytics for i4.0. After that is accomplished, all the primary use case requirements will be fulfilled. However, this system exists solely to respond to the use case needs, which are still in development, which means new requirements could appear. If new requirements appear, the communication system should suffer alterations, and for that reason, it is far from being complete.

Another proposition for future work is to allow the REST API to transcend the Augmanity use case requirements and become a service to be used for every third-party software in the company. Because of the system's modular architecture, new controllers can be added, and old ones can be edited without interfering with each other. This means that, in theory, the system could fully map every information source of the company. This implementation would eliminate the dozens of small data pipelines and substitute them with the central pipeline passing through this system. This change would not be trivial, and some improvements should be made in terms of hardware and software to handle the rising number of clients. For example, some first steps could be implementing an HMVC pattern and optimizing the system's cache.

In conclusion, the future works can be divided into two categories. First is the short-term work, which refers to fulfilling the new requirements that the Augmanity use cases should create. Second, it is a long-term work of turning the system into the interface of every software in the company.

References

[1] F. Tao, Q. Qi, L. Wang, and A. Y. Nee, “Digital twins and cyber–physical systems toward smart manufacturing and industry 4.0: Correlation and comparison,” *Engineering*, vol. 5, pp. 653–661, 8 2019.

[2] D. Santos, “Mes digital twin,” Master’s thesis, Universidade de Aveiro, 2021.

[3] M. Lopes, “Building an industry 4.0 platform: The implementation of opcon mes at avp,” Master’s thesis, Universidade de Coimbra, 2017.

[4] M. Massé, *REST API Design Rulebook*, vol. 1. O’Reilly Media, 2011.

[5] M. Ma, J. Yang, P. Wang, W. Liu, and J. Zhang, “Light-weight and scalable hierarchical-mvc architecture for cloud web applications,” pp. 40–45, Institute of Electrical and Electronics Engineers Inc., 6 2019.

[6] A. W. Troelsen and P. Japikse, *C# 6.0 and the .NET 4.6 framework*, vol. 1. Apress, 2015.

[7] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*. McGraw-Hill Education, 7 ed., 2020.

[8] M. H. Javed, X. Lu, and D. K. Panda, “Cutting the tail: Designing high performance message brokers to reduce tail latencies in stream processing,” vol. 2018-September, pp. 223–233, Institute of Electrical and Electronics Engineers Inc., 10 2018.

[9] K. M. M. THEIN, “Apache kafka: Next generation distributed messaging system,” 2014.

[10] P. Fite-Georgel, “Is there a reality in industrial augmented reality?,” pp. 201–210, 2011.

[11] E. Oztemel and S. Gursev, “Literature review of industry 4.0 and related technologies,” 1 2020.

[12] H. Lasi, P. Fettke, H. G. Kemper, T. Feld, and M. Hoffmann, “Industry 4.0,” *Business and Information Systems Engineering*, vol. 6, pp. 239–242, 8 2014.

[13] M. Ghobakhloo, “Industry 4.0, digitization, and opportunities for sustainability,” *Journal of Cleaner Production*, vol. 252, p. 119869, 2020.

[14] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson, “The industrial internet of things (iiot): An analysis framework,” *Computers in Industry*, vol. 101, pp. 1–12, 10 2018.

[15] T. H. Uhlemann, C. Lehmann, and R. Steinhilper, “The digital twin: Realizing the cyber-physical production system for industry 4.0,” vol. 61, pp. 335–340, Elsevier B.V., 2017.

[16] F. Biesinger, D. Meike, B. Kraß, and M. Weyrich, “A digital twin for production planning based on cyber-physical systems: A case study for a cyber-physical system-based creation of a digital twin,” vol. 79, pp. 355–360, Elsevier B.V., 2019.

[17] P. Fraga-Lamas, T. M. Fernández-Caramés, Óscar Blanco-Novoa, and M. A. Vilar-Montesinos, “A review on industrial augmented reality systems for the industry 4.0 shipyard,” 2 2018.

[18] K. B. Osnes, J. R. Olsen, P. Vassilakopoulou, and E. Hustad, “Erp systems in multinational enterprises: A literature review of post-implementation challenges,” vol. 138, pp. 541–548, Elsevier B.V., 2018.

[19] S. Mantravadi and C. Møller, “An overview of next-generation manufacturing execution systems: How important is mes for industry 4.0?,” vol. 30, pp. 588–595, Elsevier B.V., 2019.

[20] R. Rosen, G. V. Wichert, G. Lo, and K. D. Bettenhausen, “About the importance of autonomy and digital twins for the future of manufacturing,” vol. 28, pp. 567–572, 5 2015.

[21] A. Bratukhin and T. Sauter, “Functional analysis of manufacturing execution system distribution,” *IEEE Transactions on Industrial Informatics*, vol. 7, pp. 740–749, 11 2011.

[22] A. Pereira, “Development of software to connect cncs to nexeed mes,” Master’s thesis, Universidade de Aveiro, 2020.

[23] F. Halili and E. Ramadani, “Web services: A comparison of soap and rest services,” *Modern Applied Science*, vol. 12, p. 175, 2 2018.

[24] S. H. Toman, “Review of web service technologies: Rest over soap,” *Journal of Al-Qadisiyah for Computer Science and Mathematics*, vol. 12, pp. 18–30, 2020.

[25] M. Botto-Tobar, M. Z. Vizuete, P. Torres-Carrión, S. M. León, G. P. Vásquez, and B. Duralovic, *Applied Technologies*, vol. 1. Springer International Publishing, 2019.

[26] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, 2000.

[27] L. Li and W. Chou, “Design and describe rest api without violating rest: A petri net based approach,” pp. 508–515, 2011.

[28] X. Chen, Z. Ji, Y. Fan, and Y. Zhan, “Restful api architecture based on laravel framework,” vol. 910, Institute of Physics Publishing, 11 2017.

[29] J. Deacon, “Model-view-controller (mvc) architecture,” tech. rep., Computer Systems Development, Consulting & Training, 2009.

[30] O. A. Ragnarsson, "Importance of design patterns and frameworks for software development," 2007.

[31] N. M. Edwin, "Software frameworks, architectural and design patterns," *Journal of Software Engineering and Applications*, vol. 07, pp. 670–678, 2014.

[32] J. Richter, *Applied Microsoft.NET framework programming.* Microsoft Press, 2002.

[33] T. L. Thai and H. Q. Lam, *.NET framework essentials.* O'Reilly, 2001.

[34] A. Zainudin and A. A. Yunant, "Design an mvc model using python for flask framework development," 2019.

[35] K. Relan, *Building REST APIs with Flask.* Apress, 2019.

[36] T. Teorey, S. Lightstone, T. Nadeau, and H. V. Jagadish, *Database Modeling and Design.* Morgan Kaufmann Publishers, 5 ed., 2011.

[37] N. Garg, *Apache Kafka : set up Apache Kafka clusters and develop custom message producers and consumers using practical, hands-on examples.* Packt Publishing, 2013.

[38] J. Yu, "Exploration on web testing of website," vol. 1176, Institute of Physics Publishing, 3 2019.

[39] I. Ghani, W. M. N. Wan-Kadir, and A. Mustafa, "Web service testing techniques: A systematic literature review," 2019.