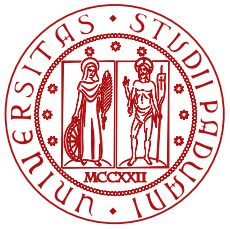

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in ICT For Internet And Multimedia

Design and Development of a Monitoring and Communication Software for Solar Panel Production Lines

Master Candidate

Merve Ofluoglu

Student ID 2073030

Supervisor

Leonardo Badia

University of Padova

Academic Year
2024/2025

To my dear sister Hanife Yucedal
and precious friends

Abstract

This thesis examines the basic principles of photovoltaic modules and solar panels and explains the working logic of the machines used in solar panel production lines. Solar panel production is a complex process that involves the synchronized operation of machines with many different tasks and technologies. The main purpose of this study is to develop a "supervisor" system that can monitor and control the entire production line from a central point. The system aims to monitor the current status of the machines, analyze messages and provide rapid feedback to events in the production line via a graphical user interface (GUI). The software system developed within the scope of the thesis is designed in a modular and scalable structure with modern software technologies. The system uses the MQTT protocol, which stands out with its lightweight structure suitable for industrial applications in the communication layer. Messages received from MQTT are listened to, parsed and recorded in the SQL Server database by a .NET background service (Supervisor.MQTT). On the user interface side, a web-based front-end developed with React and Redux technologies has been designed. This interface receives data via RESTful APIs; while listening to real-time messages from the server via SignalR. There are many features on the interface such as listing machines, unread message alerts, detailed content display with modal windows and user-specific session management. The backend has been developed with ASP.NET Core and has a multi-layered structure including controller, service, repository and entity layers. The system can successfully monitor the current status of the machines and messages in the production process with its current state. However, in the future, it is aimed to develop the system to be able to communicate not only one-way but also two-way. In this way, users will be able to control the machines directly using the MQTT protocol via the interface. This thesis reveals how a powerful and applicable solution can be developed that will contribute to the digital transformation of real-world production systems by combining modern web technologies and industrial communication protocols.

Contents

List of Figures

List of Code Snippets

List of Acronyms

GUI Graphical User Interface

PLC Programmable Logic Controller

MQTT Message Queuing Telemetry Transport

LED Light Emitting Diode

IOT Internet of Things

SCADA Supervisory Control and Data Acquisition

M2M Machine to Machine

IBM International Business Machines

CLR Common Language Runtime

FCL Framework Class Library

DOM Document Object Model

JSX JavaScript XML

MVC Model View Controller

SSE Server-Sent Event

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

XHR XMLHttpRequest

CRUD Create, Read, Update, Delete

DBMS Database Management System

ORM Object Relational Mapping

RCP Recipe

STS Status

DTO Data Transfer Object

API Application Programming Interface

DAL Data Access Layer

UI User Interface

URL Uniform Resource Locator

DoS Denial of Service

1 Introduction

1.1 Photovoltaic Systems

Photovoltaic systems[1, 2] are renewable energy technologies that convert sunlight directly into energy through the photovoltaic effect. The base of these systems is solar cells, typically made from semiconductor materials such as silicon. Sunlight hitting the surface of the cells causes electrical charges to be released, and these free charges create an electrical potential within the cell. Energy is generated as long as sun rays hit the surface of the solar cell.

While photovoltaic systems are often praised for their low operational costs and sustainability, it is important to note that they are not entirely cost-free. Regular maintenance of the solar panels, periodic cleaning to preserve efficiency, and the replacement of system components such as inverters contribute to the overall life-cycle cost. Nevertheless, compared to conventional energy sources, photovoltaic systems remain a highly cost-effective and environmentally friendly solution for long-term energy generation.[3]

To summarize the main advantages of photovoltaic systems;

- There is no need for a battery or a source to produce energy which contributes to a green environment.
- The absence of a battery also reduces operational costs.
- Easy to install.

- Can be used long term with maintenance.
- Can be installed into new or existing buildings.

There are also some disadvantages for photovoltaic systems. These disadvantages can be listed as follows:

- Requires high installation cost.
- Photovoltaic systems use sunlight as fuel and sun light is not always available which means limited energy production.
- Economical deficiency for energy storage.
- While steadily improving, the efficiency of solar panels is still low.[4]

Photovoltaic systems have an important place among sustainable energy sources due to their low environmental impact, reliability and low maintenance requirements. Not requiring any use of fossil fuels during energy production offers a significant advantage that reduces the carbon footprint.[5] These systems, which are widely preferred in industrial, commercial and individual use areas, are constantly developing in order to increase energy efficiency and promote renewable energy sources. The integration of photovoltaic technology with energy storage systems and smart grids are considered important steps to increase the sustainability of future energy infrastructure.[6]

1.2 Solar Panel Production Lines

Solar panels are devices that convert sunlight directly into electrical energy. They consist of photovoltaic cells and are used in renewable energy production. Solar panels generate electricity by being mounted on the roofs of buildings, in open areas, or in solar power plants. These panels, which take sunlight and convert it into electricity, can either provide energy to the grid or store it in batteries.

The cells used in panels are usually silicon-based and there are two main types: monocrystalline and polycrystalline. Monocrystalline cells have higher efficiency, while polycrystalline cells are more cost-effective. Solar panels do not store energy directly; the energy they produce can be stored in batteries. Battery capacity varies according to the size of the system and the need. To give an example, a 1 kW system can produce an average of 4-5 kWh of energy per day, but this depends on the amount of sunlight and the efficiency of the panel. [7]

Solar panel production lines are automated systems that start from the production of photovoltaic cells, to the processing of these cells into modules, testing, and packaging. In production lines, cells are combined with glass and protective materials to create durable panels. These production lines can consist of many machines. Each of these machines has a different task. These production lines are classified according to the energy level they produce and store.[8]

1.2.1 Working Principles of a Production Line

There are many steps in a production line for producing a photovoltaic module outlined in the following. [9]

Preparation

As a first step, the raw material that will be used in the assembly process is prepared. Additionally the glass that will be used in production must be cleaned. After cleaning, the glass-cutting machines cuts the rolls of back sheet material according to programmed production.

Stringing

The solar cells are arranged in a device known as a solar stringer, which solders a coated copper wire known as ribbon to the cell's bus bar to connect the cells in a series. The string, which is the fundamental component that forms the electrical series in the photovoltaic module, is created by this careful operation. The soldering quality is the most crucial parameter to examine. Some stringer machines are capable of managing mono and polycrystalline cells in an optimal manner, resulting in a soldering outcome that is dependable, consistent, and of superior quality.

Positioning and Bussing Soldering

After the glass has been prepped with the initial layer of encapsulant material, the strings of solar cells produced by the stringer machine are placed either automatically or manually. The equipment known as lay-up, which carries out this task in the photovoltaic module production line, may also carry out quality controls on the finished product, ensuring the soldering quality is accurate, the cells are intact, and that there are no breaks.

Once all of the strings that make up the module are positioned on the glass, you may solder the bus ribbon, which is thicker than the ribbon that connects the strings, to create an electrical circuit.

Pre-Lamination

The second layer of encapsulating material is applied when the electrical link between each string of cells is complete, and this is followed by an insulating foil known as the back-sheet. The terminal ribbon that will be linked to the junction box in a subsequent phase is brought out by the operator. Since it is still possible to fix potential problems, it is crucial to conduct some electrical tests at this stage of the production process, including an electroluminescence test, to ensure that the module is free of short circuits and damaged solar cells.

Lamination

The polymerization of the encapsulating material during the lamination process causes the multi-layer sandwich that has been assembled up to this point to become a single unit. The machinery used in this phase are laminators that were created specifically for the solar industry. These devices operate for a predetermined amount of time at a predetermined temperature under extreme vacuum. The encapsulating substance that is being utilized determines the temperature and time settings. One essential component in ensuring the product's longevity and quality is air extraction. Laminates are the photovoltaic modules that come out of the laminator. In order to facilitate a speedy procedure without waiting times, cooling systems are frequently placed after the laminator in production lines with high throughput.

Framing

The encapsulant and back sheet excess that is left around the glass after lamination is cut off to begin the framing phase, which consists of a number of tiny processes. Tape is applied directly to the laminate, or silicone is poured straight into the aluminum frame's channel to affix the frame around the photovoltaic module.

Applicaton and Last Tests

The last construction stage consists of connecting the junction box. The process is attaching the junction box to the rear sheet of the module using an appropriate silicone or glue, and forming the electrical connection between the box's wires and the bus ribbon that was created before lamination. The solar module is protected while in operation by by-pass diodes inside the box. After this is finished, the module is cleaned.

The photovoltaic module is tested and controlled when the construction process is finished. This is one of the most important stages since it provides a marketable value for the product and enables you to measure the electrical output of the module. An LED sun simulator is used to measure which can mimic certain light conditions and determine the module's peak power based on established standards.

1.3 Programmable Logic Controller (PLC)

PLC (Programmable Logic Controller), an automation device, is used to control machines or to monitor the production sections of factories. A feature that distinguishes Programmable Logic Controller from normal computers is that it has many Input/Output (I/O) inputs. Its biggest advantages include being resistant to electrical noise, temperature differences, and mechanical impacts. [10]

Each brand has its own Programmable Logic Controllers that it designs and manufactures, each with its own operating system. This controller system scans the input information very quickly and produces the appropriate output information in real time. All Programmable Logic Controllers play the role of pro-

Figure 1.1: Omron CP1E PLC (Image from OMRON)

ducing more and higher quality products in a certain period of time, with significantly lower error rates compared to traditional systems.

Data transmission is carried out by PLCs installed on specific machines on the solar panel production line. In order to create a central monitoring and control system (the purpose of this thesis), PLCs with MQTT functionality transmit these machine-generated values via specified topics to a central MQTT broker.

2

State of the Art

In this section, the problem will be explained to the reader.

2.1 Definition of the Problem

The machines in solar panel production lines currently operate independently and cannot be monitored and controlled by a central system. This situation negatively affects the efficiency and error management of the production process. It is currently not possible to obtain information about the data collected from the machines and their status, or to control them remotely.[11] See figure 2.1.

Problems can be listed as;

- Lack of Tracking the Machines:

Not all the machines have an operator checking on them but the ones that have can be monitored only locally; there is no central monitoring system that can give user an idea of all the machines' status.

- Lack of Control over Machines:

Currently, none of the machines can be controlled from a central system. This makes it impossible to intervene remotely in case of a situation.

- Lack of Communication Between Machines:

There is no communication on the network between machines, they all operate locally. This can make it challenging for machines to be informed about each other.

- Data Loss:

There is no centralized storage for data coming from machines. This may negatively affect the ability to obtain required data for reporting purposes.

- Exception Handling:

In the event of an emergency (or an error) there is no mechanism that can intervene with the error immediately from one point. Again, these processes are currently handled locally.

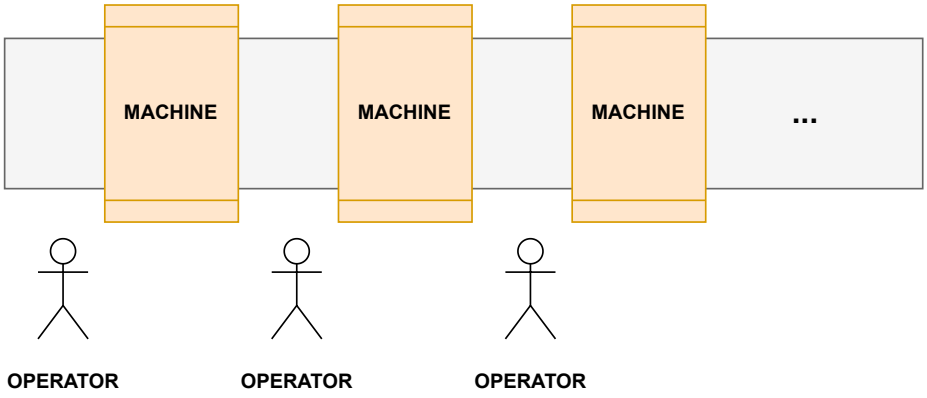

Figure 2.1: Example of a production line

To handle these problems a central monitoring and control system (it will be referred to as the Supervisor) will be implemented. A web application was deemed appropriate to solve this problem. The technologies chosen to implement this software are: .NET technology for the backend and React library for the front-end. As a messaging protocol, MQTT (Message Queuing Telemetry Transport). To better understand what should be done, these terms will be explained in the next sections. [12]

2.2 Message Queuing Telemetry Transport - MQTT

The importance of data communication has increased due to the rapid spread of the Internet of Things (IoT) and other distributed systems. Systems such as

networks and devices with low bandwidth, latency, and poor security require a lightweight and reliable communication protocol.[13] To meet such a demand, a communication protocol called MQTT (Message Queuing Telemetry Transport) was created. The basic structure of MQTT provides security and delivery assurance during data exchange, while at the same time minimizing network bandwidth and device resource usage. Due to this basic structure, MQTT is increasingly gaining importance in the M2M (Machine to Machine) and IoT (Internet of Things) sectors.[12]

The first version of the protocol was published by Andy Stanford-Clark (IBM) and Arlen Nipper (then Eurotech, Inc.) in 1999. It was initially used in oil pipe monitoring in SCADA industrial control systems. The purpose of the installation at that time was to create a protocol that was efficient and used less energy in terms of bandwidth of the devices connected via satellite connection at that time.

2.2.1 Architecture of MQTT Protocol

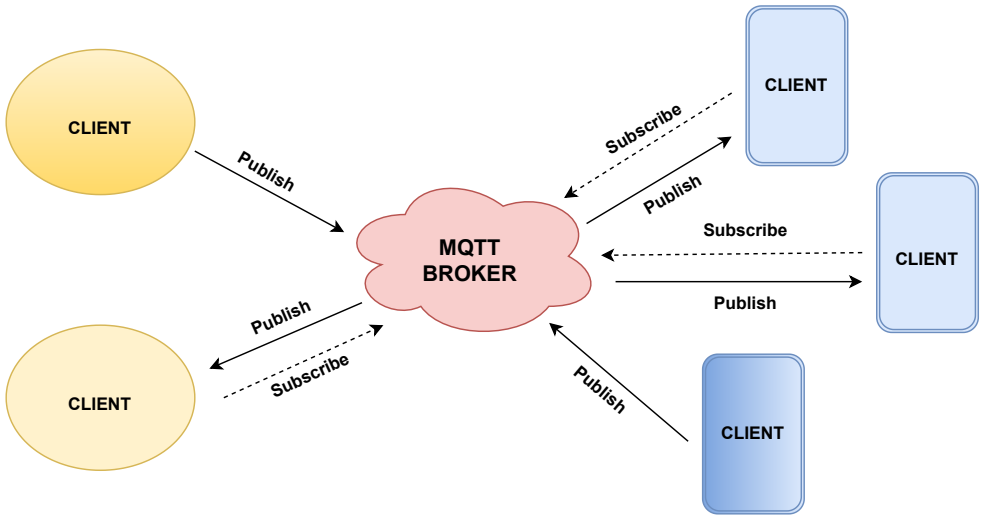

Figure 2.2: MQTT Architectural Design

While being a lightweight and simple protocol, MQTT requires a model that corresponds publisher and subscriber. Publishers produce data and send them

while subscribers subscribe to a specific topic and listen to that topic constantly. For this purpose, there are two components of MQTT protocol; an MQTT Broker to receive messages and send them to corresponding clients and clients who send or receive messages. In some situations clients can be both subscribers and publishers.

2.2.2 Important Terms for MQTT

MQTT Broker

The broker receives the messages sent by the client and directs them to the clients to whom they are intended to be delivered. The important point here is that there is no direct communication between the clients, communication occurs through a broker, it can be thought of as a kind of message pool. It is the control point that manages the communication between the clients. It also has roles in determining the clients and ensuring security. These roles can be listed as follows:

- Authorizing and authenticating MQTT clients
- Passing messages to other systems for further analysis
- Handling missed messages and client sessions

MQTT Client

Clients are devices, applications, or systems that communicate with a broker using an MQTT protocol. Clients can have two different roles; one of them is publisher and the other is subscriber. Clients in the publisher role are responsible for sending messages. These clients always send their messages via a specific topic (channel name). If the client is in the listening status, that is, receiving messages, they are called subscribers. These clients subscribe to specific topics and can only see messages from those topics. All clients generally consist of applications and devices such as power-saving or low-power IoT devices, sensors, and microcontrollers. The purpose of all these applications and systems is to provide reliable and fast data transmission through the broker. [14]

Clients are not only responsible for sending messages. While sending messages, they also manage the accuracy of the messages sent and the frequency of sending messages.[15]

Topics

The words we call topics are keywords that help organize and direct messages. A publisher always sends messages to certain topics, and subscribers subscribe to and listen to these topics. Since topics are organized in a hierarchical structure, the broker can deliver messages to the right subscribers. For example, topics such as `home/livingroom/light` and `home/kitchen/temperature` can be included in a smart home system. Thanks to this structure, messages can be managed between devices in a certain order and ease.[16]

Another important feature of topics is that they can make broad subscriptions using wildcards. For example, a subscription like `home/+/temperature` can capture messages from temperature sensors coming from all rooms. In addition, thanks to MQTT's decoupling feature, clients do not have to know each other's network addresses. This ensures that all communication takes place only through topics. This structure of MQTT becomes an effective messaging protocol in environments with many clients, such as IoT applications. Thanks to topics, each client can send and receive messages to the topics they are interested in, and this process is controlled by a central broker.

Quality of Service

MQTT's Quality of Service feature is the mechanism that determines the reliability and frequency of sending messages. MQTT offers 3 different QoS (Quality of Service) levels. Each level has a different status for the secure transmission of the message. A level is determined at the level required by the system and implementation is performed with this level.

1. QoS 0 (At Most Once): This level allows the message to be sent at most once. It does not ensure that the subscriber receives the message or not. At the same time, it does not guarantee whether the message will reach the subscriber safely. This level is ideal for situations where message loss is not critical and network traffic is desired to be kept to a minimum.

2. QoS 1 (At Least Once): This level guarantees that the message will reach the subscriber at least once. This means that if the subscriber does not receive the message, the same message will be sent multiple times. In practice, the publisher sends the message to the broker and waits for an acknowledgment. If this acknowledgment does not arrive, the publisher resends the message. This QoS level is usually used in cases where it is important that the message is definitely delivered to the subscriber. However, this situation may cause network traffic. During implementation, it should be checked whether it is suitable for systems where the message can be repeated.[17]

3. QoS 2 (Exactly Once): At this level, it is guaranteed that the message will be delivered to the subscriber only once. In order to do this, a four-stage handshake is performed. Thanks to this handshake, the highest level of security is provided and this situation increases the network load. Due to these features, this QoS level is used in critical IoT systems. Examples of these systems include medical devices and the transmission of financial data.[18]

Last Will and Testament

Last will and testament is a feature that automatically sends a message when a client or server is disconnected. In practice, a warning message is sent systemwide when a client's connection to a broker is lost. This message usually indicates that the client or broker has unexpectedly shut down or disconnected.

When a client connects to the broker, it defines the 'Last Will and Testament' message and this message is stored in the broker. If the client leaves without properly terminating the connection, the broker sends this message to a previously set topic. For example, a network break or sudden shutdown can be shown as examples of the client not terminating its connection properly. Thanks to this feature, client states are monitored and reliability is increased in critical IoT applications. [19]

Retained Message

Normally, when a subscriber subscribe to a topic, receives messages published to the topic since the time of subscription. However, thanks to the retained mes-

sage, the broker stores the last message sent by the publisher and sends this message to the subscriber who is a new subscriber to the topic. In short, retained message is a feature in the MQTT protocol that allows subscribers to receive messages immediately after subscribing to a topic.

When a retained message is sent to a topic, the broker stores this message until a new message arrives to the same topic. Thus, this retained message can be forwarded to every new subscriber.[20]

2.3 .NET Framework

A framework is a collection of structures and libraries created to support the software development process to be easier, faster, and in a certain order. It is used to make it easier for a software developer to write code on certain platforms (IDE) for a certain job. Frameworks are usually implemented for a certain programming language and make the code written in that programming language more secure and easier to maintain. The choice of which framework to use is based on the features of the project and the developer's wishes.[21]

The .NET Framework was first published by Microsoft in 2002 as a software development platform. This framework, developed to code Windows-based applications, is divided into two as Common Language Runtime (CLR) and Framework Class Library (FCL). Common Language Runtime is a virtual machine that allows applications to run, as well as providing services such as memory management, security and error management. Framework Class Library contains detailed class structures used in areas such as data management, web development, cryptography and user interface.

The latest version of .Net Framework, .Net Framework 4.8.1, was released by Microsoft in 2022. The updates that followed, starting with .NET Core and then .NET 5, enabled the platform to gain a more modern and flexible structure. Thanks to this flexibility, these new .NET versions can also run on operating systems other than Windows, such as Linux and macOS. With the subsequent .NET versions .NET6, .NET7 and finally .NET8, this different platform support was further developed, providing advanced support for performance optimizations and cloud-based applications.

2.4 React Library

React is an open source Javascript library developed by Facebook in 2011. This structure, which we can call React or React.js, is a development library created to develop user interfaces and is used quite frequently in the field of web development. In addition to helping to create single-page web applications or mobile applications, it is also suitable for complex applications when used with other libraries.[22]

2.4.1 Easy Use of React: Advantages and Limitations

Component-Based Structure

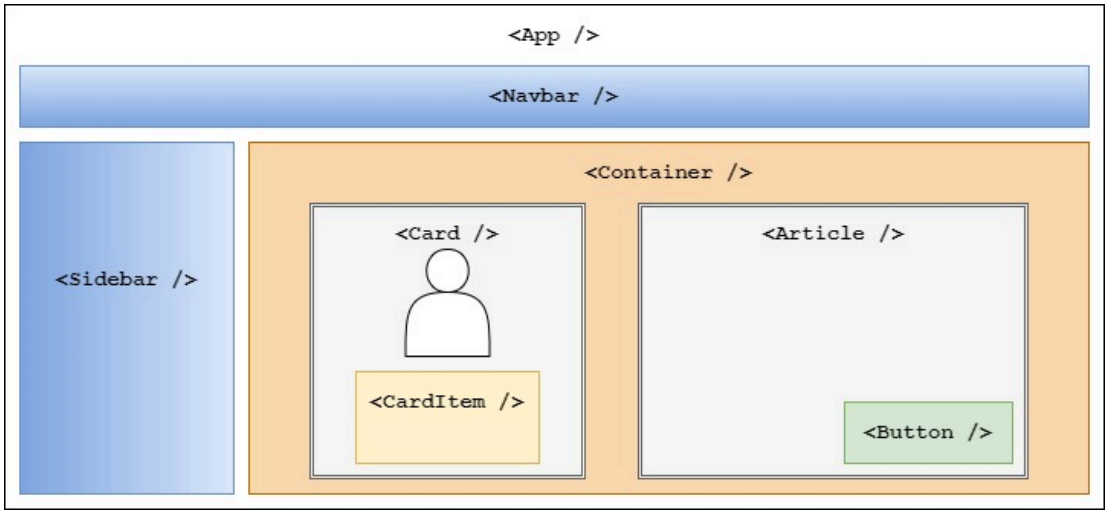

Figure 2.3: React Component Logic

React allows us to implement dynamic web applications much faster and easier than vanilla Javascript. Javascript needs more complex and large code blocks; on the other hand, React makes these blocks much more efficient, which it owes to its component-based structure. React's component-based structure allows you to take an application as a separate component down to its smallest part and code it that way. For example, everything on a web page can be a component; nav-bars, headings, each paragraph, all card structures and more. These component structures can be reused when necessary, thus creating a clean and orderly structure without code repetition. In vanilla Javascript, it is necessary to

write much more code and establish complex structure relationships to provide the same functionality.

Improved performance and Virtual DOM

Code written with React renders fast because this library uses a technology called Virtual DOM behind the scenes that increases performance. In normal web pages, when there is any change on the client or server side, the entire page needs to be reloaded in order for the change to be reflected on the page. This situation causes the re-retrieval of information that has not been updated, since it will re-sents all requests, including those made when the page was first opened, and therefore the user loses time and the web page slows down to a certain extent. However, this situation is optimized with Virtual DOM because this structure compares the previous and next states of all components and allows only the components that have changed to be updated in the Actual DOM. This technology prevents all components from being re-rendered, increasing the speed and performance of the application, especially in complex applications.

Unidirectional Data Flow

React follows a one-way data flow model. This means that data flows in a single direction from the parent component to child component. Applications developed with React.js usually have parent components and child components. The parent component passes data to the child component with a react mechanism called props, and the child component only uses this data, does not return it. This unidirectional flow makes it easier to track data and simplifies the debugging process throughout the application. If a problem occurs, it is easier to find the source of the error by tracking how the data flows through which components. This is a very important advantage in complex and large projects, because multidirectional data flow (such as in the MVC model) can often create more errors and complexity.

Small Learning Curve and Specialized Tools for Debugging

React.js is an easy-to-learn library because it is mostly based on HTML and Javascript. It has a special syntax that allows user to use HTML and Javascript together called JSX. The basic structure and component logic of React.js are quite simple to understand for beginners. It may take a little longer to understand the

more complex features of React.js (such as state management, lifecycle methods, hooks, React-Redux), but the overall learning curve is lower than many other frameworks. In addition, the learning process is supported by React's large community and resources.[23]

React offers special tools for the debugging process. React Developer Tools is a browser extension that was developed by Facebook and can be added to Google Chrome. With this tool, developers can easily examine React components through the browser, get detailed information about the states and props of each component. This feature makes it much easier to track which component processes which data, especially in complex applications, and find the source of errors. With this tool, developers can manage their applications more efficiently and save time in the debugging process.[23]

Limitations

- React.js is not a framework instead, it is a library.
- Some configurations are required to integrate with the traditional MVC framework.
- Slow development pace.
- React.js is easy to learn compared to Angular.js. However, when enriched with Redux, it becomes difficult to learn React.
- It does not use any dependency injection concept.

2.4.2 React - Redux

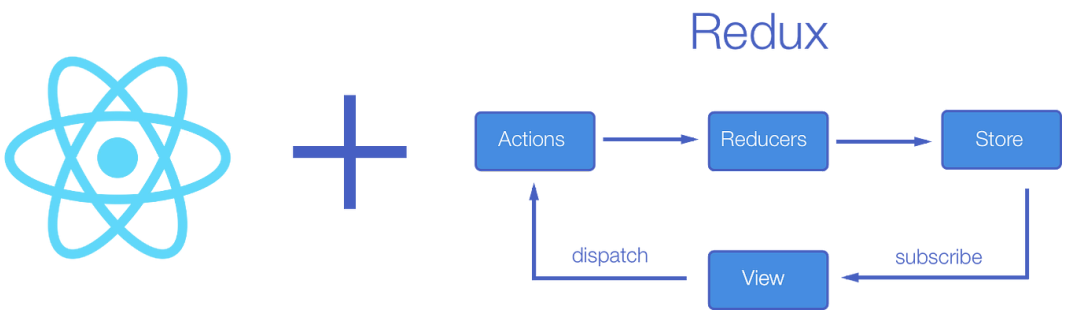

Figure 2.4: React-Redux Visual Representation (Image from Medium)

Redux is a state management library for Javascript applications. State in the Redux is the properties and information that each component the application currently has. The change of these properties and information is the definition of the concept of state. If we consider it from React, as the number of components in the application increases, it becomes very difficult to control each component and the states it contains. This situation causes uncontrolled data flow. This situation is generally called state management problems and Redux was developed as a solution to this problem by Dan Abramov, one of the creators of the React.js library.[24]

There are certain structures in Redux that perform state management, one of them is the store structure.

Store: This is the structure that Redux creates to keep all states in the application in a single and central place. Because of this structure, all states are controlled in a single place and it becomes much easier to manage. Store allows all components in the application to access the desired state without ”parent” - ”child” control, it allows to subscribe and listen to this state. Therefore, when there is an update in the state, the components that are subscribed (listening) to this state are rendered.[24]

Another Redux structure established for state management is actions.

Action: These are structures that describe any event that occurs in the application. They are simple Javascript objects with a type field. They are structures that carry data between the application and the store. The data they carry is called payload. When calling an action, it is mandatory to pass the type parameter. Because the application finds the action to be processed according to the type parameter passed and passes the payload there. All parameters except the type parameter are optional.[24]

The last structure we will define to complete state management is reducers.

Reducer: This structure is passed as pure functions in Redux. What it does is handling state logic. It takes an initial state and action type as parameters. According to these parameters, it updates the relevant state and returns the new state. Since it is now an updated state, all components listening to this state are

rendered.[24]

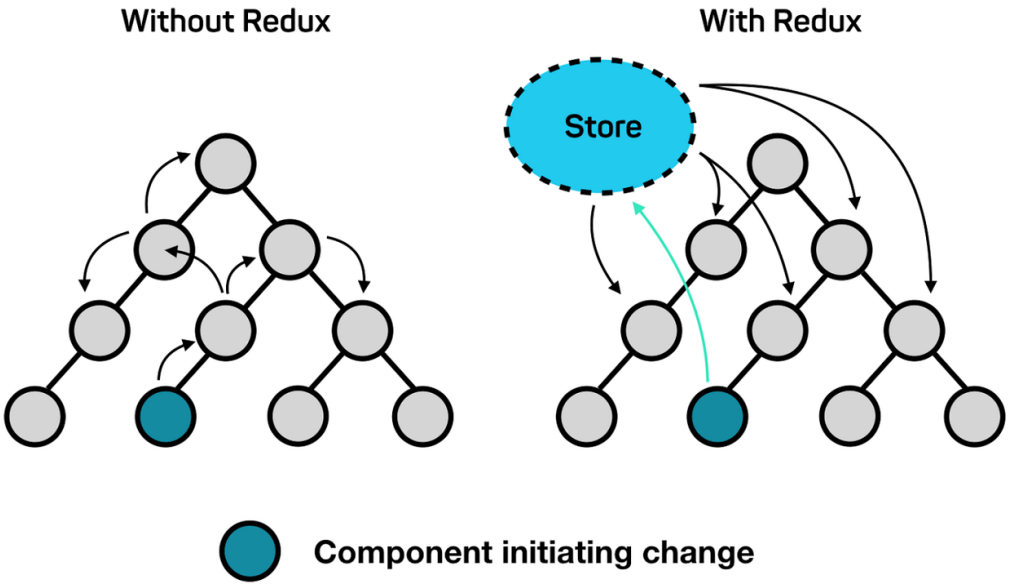

Figure 2.5: Redux State Management Visual Representation (Image from Medium)

2.5 SignalR

SignalR is a real-time web application library that runs on ASP.NET technology. SignalR provides two-way and continuous communication between the server and the client. It is frequently used in applications that require real-time data (chat applications, live notification systems, instant data updates, etc.). SignalR allows the creation of long-term connections between the client and the server and communicates with different methods such as WebSockets, Server-Sent Events (SSE) and Long Polling. Which communication protocol to use is determined automatically based on browser and server capabilities.

Features of SignalR

Real Time Communication: SignalR provides a continuous connection between the server and the client, so it can send instant data updates to the client.

Figure 2.6: SignalR Communication of Client and Server

Auto Protocol Selection: SignalR uses WebSockets if there are browsers that support WebSockets. If this is not supported, it provides the connection using alternative methods.

Bidirectional Communication: With SignalR, the server can send data to the client, and the client can send data to the server, making it ideal for dynamic and interactive applications.

Group Communication: SignalR can organize clients into groups and send data to specific groups, which is useful for bulk notifications.

2.5.1 Hubs

SignalR provides communication between the client and server using a high-level pipeline called a hub. Hubs enable the client and server to call each other's methods. SignalR automatically routes this communication between client and server, and data can be sent and received with strongly typed parameters. This makes model binding possible.

SignalR offers two built-in hub protocols: a JSON-based text protocol and a MessagePack-based binary protocol. MessagePack creates messages with smaller size compared to JSON. However, for MessagePack support, older browsers must have an XHR level of 2.

Hubs send messages by calling scripts on the client side. The message contains the name of the called method and its parameters. Method parameters are serialized according to the selected protocol and deserialization is performed by finding this method on the client side. Afterwards, deserialized data is passed to the appropriate method and executed.

2.5.2 .NET and SignalR

SignalR is a library that works with the .NET framework. Therefore, it offers an ideal solution for real-time communication in .NET applications. It does this by establishing bidirectional data exchange between the server and the client. The structures used in this communication are the structures called Hubs mentioned in the section above. The fact that hubs allow servers and clients to recognize each other's methods ensures that data exchange is established without interruption.

Again, the protocol supported by SignalR, which provides seamless integration with .NET, is WebSockets, but it also works with alternative protocols such as Long Polling or Server Sent Events for older browsers. In order to increase performance in this bidirectional communication, it uses structures such as JSON or MessagePack mentioned above.

2.5.3 React - Redux and SignalR

SignalR and React.js are compatible with each other to control real-time data exchange. While SignalR provides instant data transfer from server to client, React updates its state with the received data and due to the updated state, all components in the application that listen to this state are updated instantly. This takes the application to a more advanced level in terms of speed, performance.

Redux, which acts as state management, is fully compatible with SignalR, which provides instant updates. By storing the real-time data received by Sig-

nalR in Redux's store and updating the necessary components, a balanced and uninterrupted data exchange takes place in every corner of the application. In theory, in an application, messages received with SignalR are dispatched with Redux actions and stored in the store, and this updated data is passed to all relevant React components that subscribe to the store. Using these three technologies together provides a powerful solution, especially for applications that require instant updates (such as chat, live data streams, notifications). Since the problem we will solve requires instant data display from machines, these technologies were deemed appropriate.[25]

3 Implementation

3.1 Project Setup

3.1.1 N-Tier Architecture

In software projects, different architectural patterns are used to increase the functionality and modularity of the application. These patterns are selected according to the size of the project, its requirements and long-term maintenance needs. Examples of these are; Monolithic Architecture, N-Tier Architecture, Micro-service Architecture, Event-Driven Architecture, Server-less Architecture. Each of these has its own advantages and disadvantages. To briefly define:

- Monolithic Architecture: A type of architecture where all components of the application are located in a single code base.
- N-Tier Architecture: The application is divided into layers with different functionalities. Each layer has a specific responsibility and communicates with other layers through interfaces.
- Micro-service Architecture: The application consists of small services that work independently of each other and perform a specific function.
- Event-Driven Architecture: The application consists of independent components that communicate through events. For example, events occurring in a system trigger other components.
- Server-less Architecture: The application is developed with the functions offered by cloud service providers without dealing with server configuration.

Figure 3.1: Supervisor Solution Projects

Since the Supervisor project is a large-scale and long-term software development project, a layered architecture was preferred. This preference was made for the reasons explained below:

1. Modularity: Each layer in the Supervisor project is designed for a specific functionality, these layers are:

Presentation layer (React): Interaction with the user occurs in this layer. The user makes a request with a button or similar tool. While the request is transmitted to the Business Logic Layer, it transmits information and warnings from the lower layers to the user (Supervisor.client and Supervisor.Server).

Business Logic Layer: It is used to perform additional checks outside the database. Depending on the control result, either the request is transmitted to the Data Access Layer (Supervisor.service and Supervisor.MQTT) or warnings are transmitted to the user regarding the request result.

Data Access Layer: It is the layer that performs the operations by transmitting the requests from the Business Logic Layer to the database or databases it is connected to and transmits the result back to the Business layer. All database connection and operation commands (CRUD) are written and developed in this layer (Supervisor.data and Supervisor.dto).

This modular structure allows each layer to be developed and tested independently.

2. Maintenance: The N-Tier Architecture makes it easy to add new features to the project or update existing features. For example: If a new user interface needs to be added or updated, only Supervisor.client is modified. If a new database needs to be added, this only requires work on Supervisor.data. Such a structure makes the project more flexible and manageable in the long run.

3. Real-time Communication and Technological Integration: This project brings together different technologies such as React, SignalR, and MQTT. The

layered architecture makes it easy for these technologies to work together: SignalR is used for real-time communication and provides the connection between React and the server. MQTT provides the processing and management of messages from devices. This structure allows the technologies to be optimized independently of each other.

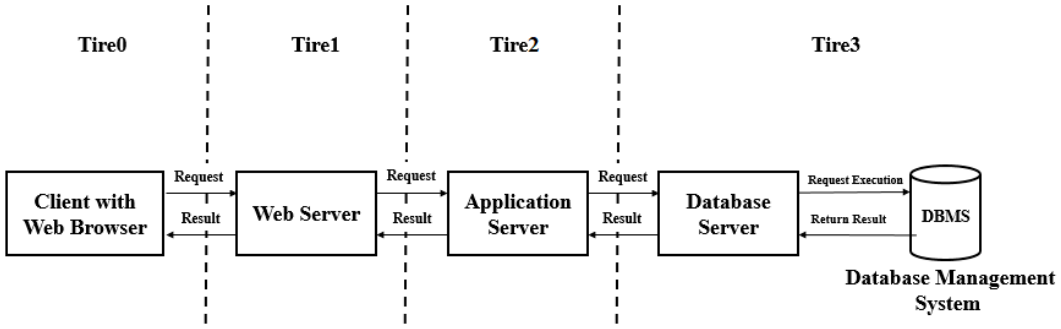

Figure 3.2: N-Tier Architecture

Now we will examine the Supervisor implementation project by project.

3.1.2 Data Access Layer

This layer corresponds to `Supervisor.Data` project in our system and manages database operations using entity framework core.

Entity Framework Core

Entity Framework Core is an ORM (Object Relational Mapping) tool developed by Microsoft and can be used with .NET Framework and .NET Core.

Why Entity Framework?

- There are two opposing approaches called DatabaseFirst or CodeFirst, these approaches provide automatic creation of the data layer. If the DatabaseFirst approach is used, the entities (class structures) of the tables created in the database are created in the code, but in the CodeFirst structure which is the opposite of DatabaseFirst, the entities with defined class structures provide the creation of table structures in the database.

- Entity Framework makes performance improvements when creating and running queries, so database operations are faster.

- Entity Framework supports different database platforms, so you can work on different databases using the same code.

Entity Framework performs mapping operations through its own special objects, some of these objects are defined below;

DbContext: It is the most important object of Entity Framework, it manages all objects related to the database.

DbSet: It represents the equivalent of a table in the database and is used to perform CRUD operations.

Entity: It is the objects corresponding to the tables in the database.

Mapping: It defines the relationship between Entity and database tables.

Command: It is used to manipulate the data in the database.

Query: It is used to query the data in the database.

Now first we add necessary entity framework core packages using NuGet or .Net Command Tool; `dotnet add package Microsoft.EntityFrameworkCore` after this command we can use ef-core in our project and we start by adding the entities to the project followed by dbContext. After all entities and their corresponding DbSets are defined on the context, repositories are written.

Repository: Since software applications usually have structures that perform database operations intensively, this structure, which allows us to perform these operations in a more practical way at once within the framework of the reusability principle, instead of writing the necessary database operations over and over again at each point of the relevant application, is called a repository. In our case an interface with generic functions has been implemented as base repository. Since Supervisor project has a lot of entities with different parameters, a generic class has been used to prevent code-rewriting.

Generic Classes (<T>): Generics are classes and methods that have placeholders (type parameters or parameterized types) for one or more types. This lets programmer to design classes that take in a generic type and determine the actual type at run time. A generic class encapsulate operations that are not spe-

cific to a particular data type.

With this, all the repositories that derive from this base repository will have generic methods and and will use these methods with respect to their own data types.

For each machine that is an entity, there are two message types; STS and RCP, in other words a recipe message and a status message.

While STS message has a lot of parameters including machine type and status, this message is considered as a heartbeat message. MQTT Broker recevies this message almost every second to let user know tho status of the machines, if the machine is on or off or in another state, could be in an error state. On the other hand RCP messages contains Recipe of a solar panel that is currently being processed.

Recipes:
A recipe in solar panel production line is contains general information of the glass and panel to be created. In MQTT Broker recipe data is sent as a JSON array:

```
{
  "RECIPE": [
    {
      "Recipe Name": "FoilLenght: 0 mm; PunchingPos: 0 mm;
    ReleasePos: 0 mm;"
    }
  ]
}
```

Code 3.1: Recipe JSON example

Overall for all machines; STS and if exist RCP entities have been created as well as DbSets in context and Repositories that derives from a base repository, which also derives from main interface IRepository.

Similarly `Supervisor.Dto` has entities that correspond to real entities created in the database and user defined entities that have no actual table reference.

Data Transfer Objects

The reason for using DTO (Data Transfer Object) in software development is data transfer. For example, when a client application needs to receive data from a server or send data to a server, DTOs help to carry out the data transfer process effectively and regularly.

DTO objects are indispensable for our API-based developments.

Exposing entity classes to the outside world through APIs is a security vulnerability also using entity classes in the controller and service layer may cause some errors.

Some fields in the entity class; these (log-like fields such as createdAt, createdBy, updatedBy) may concern the internal model or database of the application, so unnecessary fields should not be sent to the client side unnecessarily.

Database objects (entities) are not passed to the business logic layer, DTOs are used instead. This reduces the tight dependency between layers and allows easier management of changes.

This structure, as complete, creates Data Access Layer.

3.1.3 Business Logic Layer

In Supervisor project, Business Logic Layer consist from 2 projects; `Supervisor.MQTT` and `Supervisor.Service`. These 2 projects will be handled separately.

`Supervisor.MQTT`

Supervisor.MQTT is an important component in the Business Logic Layer of the Supervisor project and functions as a module that provides communication between devices using the MQTT protocol and works as a background worker in the project. This module listens to messages from machines in the production line that sends messages to MQTT Broker, processes them, and routes them to other parts of the system. It also optimizes the routing and processing of messages by supporting an event-driven structure called event-bus. To create the routing mechanism, event-bus structure uses its handlers.

Background Workers

`BackgroundWorker` is used to perform long-running or resource-intensive operations (for example, data retrieval, complex calculations, or large file operations) without occupying the main thread (UI). This allows the application to remain responsive to user interactions and provide a smooth user experience.

In the Supervisor project, a continuous data saving to a database is performed by connecting to the MQTT broker, in this case, the MQTT implementation is defined in a background worker to prevent the database and the system from locking in a possible situation.

Event-Bus

Event-bus is a communication mechanism that directs events from one component to another in software systems. It supports the principle of loose coupling, allowing a component to send messages without establishing a direct dependency on another component. Event Bus allows events to be processed by one or more ”handlers”.

Event-bus, used in the Supervisor.MQTT project, associates incoming messages with a specific ”topic” and directs them to the ”handler” function defined for this topic. It is used to organize messaging within the system.

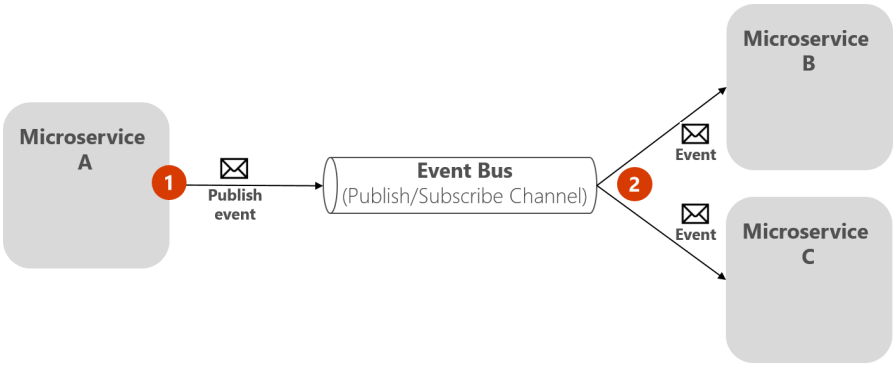

Figure 3.3: Event-bus structure. Image from Microsoft Learn

Event-Bus Interface

Event-bus interface has two methods called `Subscribe` and `PublishAsync`:

- Subscribe Method: Subscribes a handler function/class to the event-bus for a specific topic.

  - A `ConcurrentDictionary` named `_handlers` is used. This dictionary stores a handler function for each topic.
  - If a handler for the same topic already exists, an exception is thrown.
  - Lastly a method tries to add the given topic and handler to the dictionary. If the topic has already been added, an exception is thrown.

- Publish Method: Publishes a message for a specific topic and runs the handler that subscribes to that topic.
  - The given topic is searched in the `_handlers` dictionary.
  - If a handler is found, the message parameter is passed to this handler and the handler is run asynchronously.
  - If there is no handler for the relevant topic, an exception is thrown.
  - Another method tries to find a handler for the given topic. If the handler is found, the handler is called and the message parameter is passed.
  - If the handler does not exist, an exception is thrown.

After event-bus implementation, events are connected to handlers, but before that, handlers must be defined.

Handlers

The event-bus structure used in the Supervisor.MQTT project enables the forwarding and processing of messages in the system using handlers which are structures that contain a specific business logic, parse incoming messages and, if necessary, integrate with other services and save them to the database or perform other operations.

A `MachineHandler` receives incoming messages (for example, messages in JSON format from MQTT), processes them, routes them to appropriate services and saves them in the database. The purpose of this class is to parse and process messages appropriately in order to fulfill a certain business logic. Every `MachineHandler` has two methods:

- `_stsService`: A service responsible for processing STS (Status) messages and saving them to the database. This service is used to save or update machine status information using necessary repositories that explained in the Data Access Layer of this thesis.

- `_rcpService`: A service responsible for processing RCP (Recipe) messages and saving them to the database necessary repositories that explained in the Data Access Layer of this thesis. This processes and saves recipe data.

Both services are injected into the handler class with the Dependency Injection (DI) method.

After constructor and necessary injections, two methods have been implemented; `HandleSTSMessage` and `HandleRCPMessage`

- `HandleSTSMessage`: Processes an incoming STS (Status) message
  - Using `JSONConvert.DeserializeObject()`, the message is converted from JSON format to a machine entity.
  - If the message cannot be deserialized, no action is taken.
  - Then relevant method adds the STS object to the database or updates the existing record if there is already a record with the same `MachineId`.
  - A conditional check is used to determine which record to update.

- `HandleRCPMessage`: Processes an incoming RCP (Recipe) message.
  - Using `JSONConvert.DeserializeObject()`, the message is converted from JSON format to a machiine entity.
  - If the message cannot be deserialized, no action is taken.
  - Then relevant method adds the RCP object to the database.

More methods can be added here as the handler needs before sending messages to the related services; for example if a parsing operation is needed, JSON message that came from event-bus must be parsed before sending it to the service.

These handlers are implemented for all relevant RCP and STS message entities.

MqttHandler

In the MqttHandler section, the bridge between MQTT and event-bus is established.

```
1 public class MqttHandler : IMqttHandler
2 {
3     private readonly IEventBus _eventBus;
4
```

```
5       public MqttHandler(IEventBus eventBus)
6       {
7           _eventBus = eventBus;
8       }
9
10      public async void HandleIncomingMessage(string topic, string
        payload)
11      {
12          if (!string.IsNullOrEmpty(payload))
13          {
14              await _eventBus.PublishAsync(topic, payload);
15          }
16      }
17  }
```

Code 3.2: MqttHandler class implementation

- The `MqttHandler` class has an `IEventBus` dependency to route messages over the event-bus.
- An instance of `IEventBus` is injected into the class using Dependency Injection.
- The `HandleIncomingMessage` method receives an incoming MQTT message, checks its content, and sends it to the event-bus. If the payload is not valid, no action is taken. If the message is valid, the `PublishAsync` method is called to forward the message to the event-bus.
- The `_eventBus` routes the message to the appropriate handlers.

The `IMqttHandler` and its implementation, the `MqttHandler`, provide an effective structure for processing incoming MQTT messages. This class checks the messages and forwards them to the appropriate handlers using Event Bus. The structure has been made as asynchronous and modular to provide great advantages in terms of performance and extensibility. In the Supervisor project that requires real-time data processing this structure makes it easier to manage message traffic.

The last part of this mechanism has been implemented in the Program.cs file; where all the declarations and bindings have been made.

Program.cs

Program.cs establishes a relationship between event-bus and handler structures, ensuring that messages coming to specific topics are processed by the correct handlers. This process enables the system to be structured in a modular

manner for processing messages. In this section, all repositories and services should be defined before moving on to the binding section. After the services are defined with their life-cycles, the binding section can be implemented.

MqttWorkerService

This background service has two elements; the constructor where all parameters have been defined and injected, and an `ExecuteAsync` method where the message from the Broker has been received and processed.

The `MqttWorkerService` derives from `BackgroundWorker` class and implements a constructor that receives some parameters but for this case most important one is `IMqttHandler` that is explained before.

The constructor, creates client and options to secure a connection with the broker. To reach that goal necessary broker and port informations have been fetched from `appsettings.JSON` file. When this is implemented, the client now is ready to connect to broker and subscribe to topics. This section is handled with `ExecuteAsync` method.

The`ExecuteAsync` method runs continuously when the service is started, connects to the MQTT broker, subscribes to specific topics, and listens for incoming messages and makes them ready to process.

The method uses a `CancellationToken` to ensure continuous operation. This token allows the service to exit this loop when it is wanted to be stopped. The loop works with a conditional statement. Thus, the operations continue as long as the service is open.

First, the MQTT client is checked. If it is not connected to the broker, the a connect method is called to establish the connection. During this connection, settings such as the broker address and connection port are applied using MQTT client options. If the connection is successful, the service determines the topics it should subscribe to and subscribes with a subscribe method. This topic information is taken from the configuration in the `appsettings.JSON` file. The code ensures that the MQTT broker connection and topic subscriptions are configured properly.

If an error occurs during the connection or subscription, this error is caught in the catch block. This ensures that the system does not crash and retry in the event of a connection failure. The error message is logged as well. The code uses `Task.Delay` to add a 1-second delay between connection attempts. This prevents the system from increasing its resource consumption by trying to connect too often.

Finally, an event is defined for the MQTT client to listen for incoming messages. This event is triggered when a message arrives from the MQTT broker. The message content (payload) is decoded in UTF-8 format and read as a string. The message is then processed by another method. This method retrieves the topic and content of the message and directs it to the relevant processing logic. Thus, each incoming message is handled by an appropriate handler. When the message is sent to the appropriate handlers for data insertion, it is also sent to the client side using `HttpClient`.

`HttpClient` has the information of message controller. On the message controller there is an end-point that will receive this message. When received, it will broadcast this message to the all channels using SignalR.

As a conclusion, this structure ensures that the system remains connected to the MQTT broker at all times. If the connection is lost, reconnection attempts are made and the service continues to operate. Message listening is initiated immediately upon connection to the broker and incoming messages are processed appropriately. After that it will ensure message has been broadcasted to listening channels. This system provides an infrastructure optimized for real-time messaging needs.

3.1.4 Presentation Layer

Presentation Layer consists of 2 projects; `Supervisor.Server` and `Supervisor.client`. These two projects together creates the presentation layer and act as a bridge between the user interface and the business logic layer. Supervisor.Client is a front-end application developed using React and React-Redux on the other hand Supervisor.Server is a back-end application that works as a Web API that handles user requests from front-end and sends them to relevant services. Communica-

tion between these two projects is provided by HTTP requests (REST API) and SignalR technology.

Supervisor.Server

Supervisor.Server is a back-end application that works as a Web API and communicates with the React-based front-end (`Supervisor.Client`). This project offers extensive functionality by providing real-time data communication via both REST API and SignalR. The project includes many important features, from session management to machine and message operations.

Authentication

Project starts with a login page for this purpose a login interface in the front-end and an authentication controller in the back-end has been implemented. This controller defines a Login API endpoint and performs the user login process. This endpoint, specified with `[HttpPost("login")]`, receives and validates the login information from the user.

The username and password information from the client side is compared with the username and password that present in the `appsettings.JSON` file. If the username and password matches, the user can successfully login. The username information of the logged in user is recorded in the session using session mechanism. Then, an object containing the session information is created and returned to the client with a 200 OK status.

If the username and password does not match, a response is returned to the client side with a 401 Unauthorized status code and a message "Invalid username or password". This structure provides a basic validation mechanism for both managing user sessions and ensuring the security of the API.

Similarly, on log-out request, session info is cleared thus results with routing user to login page.

Machine and Message Controllers

`MachineController` is a controller that manages operations related to machines in the `Supervisor.Server` API. This controller provides basic operations

such as listing, viewing details, creating, updating and deleting machines. It also provides more comprehensive information by querying the number of unread messages associated with machines.

One of the most basic functions of `MachineController` is to list all machines in the system. A method that retrieves the available machines from the database and returns to the client. If an error occurs during the operation, a 500 Internal Server Error response is returned along with the error message. Similarly, another method retrieves the details of a specific machine. If the requested machine is not found, a 404 Not Found response is returned to the client.

`MessageController` is a controller that provides message management in the `Supervisor.Server` API and communicates with both clients and other components via the SignalR hub. This controller allows receiving messages associated with machines, querying unread messages, and updating message statuses. In addition, it broadcasts messages to clients in real time using SignalR.

One of the main functions of this controller is to return all messages belonging to a specific machine in response to requests from clients. Thanks to this function, users on the client side can easily access the message history associated with a specific machine. In addition, another endpoint that returns only unread messages allows users to quickly see which messages have not yet been processed. This is especially useful in managing operational message processes.

Another important function is to mark messages as "read". Users can mark all messages collectively or a single message as read. During these operations, the statuses of the relevant messages in the database are updated and synchronization with clients is ensured. This allows centralized management of message statuses and allows users to regularly monitor message traffic.

Finally, the controller uses SignalR to broadcast an incoming message to all clients. This broadcasting process enables real-time communication between clients. For example, when a new status message arrives for a machine, this message is transmitted to all connected clients via SignalR. In this way, user is instantly informed and coordination is achieved throughout the system.

`MessageController` offers a comprehensive solution for message management. Functions such as receiving messages, querying their unread status, and forwarding them in real time are the basic features of the controller that increase user experience and system efficiency. At the same time, real-time communication provided by SignalR strengthens the dynamic structure of the system, allowing users to access up-to-date information at all times. All these functions create a powerful and extensible structure thanks to the tight integration of the controller with the service layer.

SUPERVISOR.CLIENT

`Supervisor.Client` is a web application that creates the user interface of the Supervisor system and is developed with the React library. In the `App.jsx` file, which is the basic entry point of the project, page orientations (routing), general style components, user login screen and different pages separated according to user roles are defined. React's functional component structure and the react-router-dom library are used to direct users to specific URLs. For example, the Login component corresponding to the "/" path is the interface that performs user authentication. In addition, the "/admin" route represents the `AdminDashboard` component, which is the admin panel, "/mainpage" represents the screen of production line operators, and "/admin/machinelist" represents the component that provides the list of machines.

The react-toastify library has been integrated for notifications that can be used globally within the application. `ToastContainer` is configured to be usable on all screens of the application and provides instant notifications to the user about system events (for example, when a new message arrives, when the machine status changes, etc.). In addition to this, a modern and user-friendly interface is provided with style and component libraries such as Bootstrap.js, Bootstrap.css and Slick Carousel which both used to display data. And all these components are centrally organized in App.jsx.

One of the important technical details is the management of the SignalR connection within the application. Using the `useEffect` hook, the `SignalRService` component is executed once when the application is first loaded and a real-time WebSocket connection is established with the server. Because of this structure, messages sent by the server side can be listened to instantly on the client side.

The SignalR connection is managed in accordance with the component lifecycle; that is, the connection is terminated when the user leaves the page.

After the connection is successful, an event called `ReceiveMessage` is defined to listen to messages from the server-side. When this event is triggered, the message reaches the client and a notification is displayed to the user on the application. The incoming message is displayed in the upper right corner of the screen for ten seconds via the `toast.info()` function. This structure is a very effective method to instantly inform the user when an event occurs in the system and to include them in the process. In addition, thanks to the customizability of these notifications, it is possible to present critical messages in the system in a more visually appealing way.

A function called `stopConnection` has also been defined for the system to work reliably. This function is called when the user leaves the system or the application closes and terminates the active SignalR connection. This not only terminates the connection, but also makes the connection object null and ready to be restarted. With this method, connection management is carried out in a controlled manner, resource usage is optimized and the client is prevented from creating multiple connections.

Login and Logout Processes:

In the Supervisor application, user login processes are designed by combining React components and Redux state management. The `Login.jsx` component, which is the entry point of the application, receives the username and password information from the user and sends this information to the `loginUser` action defined via Redux. The user form is managed with React state using the controlled input structure. When the form is submitted via (`handleSubmit`), the `dispatch(loginUser(...))` call is triggered. With this process, an asynchronous login process is started on Redux and the user interface is updated according to the status of the login process.

When the login process is started, the state in the auth reducer is updated and the loading flag is set to true. This allows the user to be shown a “loading” animation. If the login is successful, the user information (username, sessionId)

returned from the server is saved to the Redux store and simultaneously written to `localStorage`. In this way, the user session becomes permanent in the browser. As soon as it is understood that the login process is successful via Redux, the `useEffect` hook is activated and the user is automatically directed to the `/mainpage` route. This process is handled with the `navigate()` function and the user is directly directed to the home page and taken into the application.

If an error occurs during the log-in process, the `LOGIN_FAIL` action is triggered, and the error message is written to the `loginError` in the reducer. This situation is also checked within the same `useEffect` hook and a visual error message is displayed to the user using the react-toastify library. Thus, both successful and unsuccessful session attempts are directly notified to the user. This structure plays an important role as the basic feedback mechanism that manages user interaction.

The `authReducer` defined on the Redux side is a state manager that manages all login and logout operations. Situations such as `LOGIN_REQUEST`, `LOGIN_SUCCESS`, `LOGIN_FAIL` are processed through this reducer. When the login is successful, the `userInfo` on the system is updated and the system recognizes the user's identity. For logout a function is implemented and this function sends a request to the server to end the user session and clear the localStorage. Then the Redux state is reset to its `initialState`. This completely logs the user out of the application and redirects them to the form screen for the next login.

As a conclusion the login and logout processes in the Supervisor application are designed with the combination of React components and Redux's state management structure. This structure dynamically shapes the user interface according to the login state and provides secure session management by working synchronously with the server side. Users' logins to the system are made sustainable by storing and reusing session information in `localStorage`. This approach is structured in line with the standard session management techniques used in modern front-end applications.

Main Page

After completing the login process in the Supervisor application, the user is directed to the MainPage component, which is the main screen of the system. This screen is the most basic monitoring interface that the application offers to the user. Due to the nature of the application, the status of the machines in the system and the messages belonging to these machines are displayed through this main component. In the React architecture, the MainPage works as a container component that organizes the high-level structure of the application. The interface structure starts with a fixed Navbar at the top; This bar usually contains basic information such as the application logo, username or session transactions. A `NavigationMenu` component located just below the Navbar is a horizontal menu that allows switching between modules in the application. This structure greets the user with a simple interface, avoids unnecessary complexity and directs the focus of attention to the main functions.

The rest of the page contains an area that dynamically displays content according to the user's selection. This content is determined by the `mainComponent` variable controlled via the Redux store. For example, if the machine information is loaded into the relevant Redux state, the `MachineCarousel` component is displayed in this section. This component is the section where all machines are presented in a horizontally scrollable structure as visual cards. Each card contains basic information about the relevant machine and the number of unread messages. In this way, the user can quickly evaluate the status of the machines in the production line.

As a result, the main page component forms the basic view of the Supervisor application and acts as the central control panel of the system. The user can easily monitor the current status of the machines in the system, the number of unread messages and their details through this screen. This structure allows the application to operate as a simple but functional surveillance system. The Redux infrastructure running in the background ensures the reusability of the components and the achievement of a consistent data flow throughout the application.

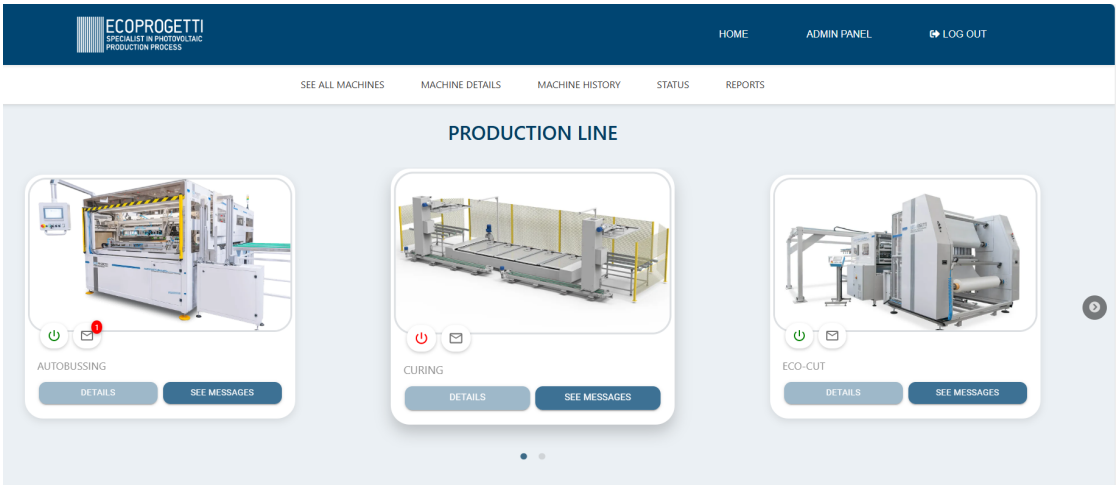

Figure 3.4: Main Page of the Application

In the Supervisor application, one of the basic components used to visually present the status of the machines in the production line to the user is the `MachineCard` component. This component is rendered individually for each machine in the `MachineCarousel`, a horizontally scrolling structure on the `MainPage` screen. Each card serves as a unit that summarizes the basic information about a machine and provides the user with fast and intuitive information about the current status of the relevant machine. The `MachineCard` includes the name of the machine, the number of unread messages belonging to that machine, and sometimes status indicators. Unread message information is usually presented as a distinct badge, drawing the user's attention to newly arrived or unprocessed data. In this way, monitoring the machines in the system is supported not only numerically but also with a visual focus.

When the user hovers over this card, the card comes to the fore with a visual growth animation (hover effect), which is an important interface detail that increases user interaction.[26] Thanks to the clickability of the card, the user can open a modal window to view detailed messages belonging to the relevant machine. This modal allows the user to instantly read messages belonging to the machine, mark them as read, and interact with the system. `MachineCard` is the first point of contact that initiates this interaction and offers both an informative and interactive structure in terms of user experience. At the same time, the `machinesData` object coming from Redux is used as the basis for the creation of these cards, so the interface remains constantly synchronized with the current

information coming from the back-end. As a result, `MachineCard` is not only a visual representation in the Supervisor system, but also an interactive component that initiates a two-way flow of information between the user and the system. Thanks to its simple but effective structure, it is easier for users to manage, monitor, and intervene in machine traffic in the system.

Figure 3.5: Machine Card component

In the Supervisor application, the status of each machine is presented to the user through the specially designed `MachineCard` component. This card, as seen above, contains rich content in both visual and functional terms. The high-resolution machine image in the center of the card makes it easier for the user to recognize the physical production line in a digital environment. The icons placed in the lower right corner just above the image symbolically represent the current status of the machine. For example, the green power icon clearly indicates to the user that the machine is active and running. Such status indicators increase the traceability of the system in real time.

Two buttons at the bottom of the card allow the user to access more detailed information about the machine. When the ”DETAILS” button is clicked, a modal window containing technical and structural information about the machine opens. This modal will show user details such as the model, status, and unread messages of the machine. On the other hand, the ”SEE MESSAGES” button triggers a separate modal showing the received system messages from that machine. Here, unread messages are particularly emphasized. As seen in the image, there is a red notification badge in the upper right corner of the envelope icon, indicating that there is a new message for the relevant machine. This structure informs the user about the current developments in the system and facilitates rapid action.

The `MachineCard` component is not only a visual representation in the Supervisor interface, but also a data-driven control center. Through this card, the user can see the current status of the machine and easily access detailed information or system messages specific to that machine. This structure increases the interactive power of the system by making the monitoring of the production line both accessible and user-friendly.[10]

The `MachineCard` in the Supervisor application, offers the user two actions: ”DETAILS” and ”SEE MESSAGES”. Both actions are designed to provide the user with more detailed information through a modal window. These modals are structured as React components and open without disrupting the current state of the page at the moment of clicking, providing the user with a fast and seamless experience.

The first modal opens when the ”DETAILS” button is clicked. This modal summarizes the physical and operational status of the relevant machine in a simple way. For example, the name of the machine, its current operating status (e.g. ”running”), and the number of unread messages are presented to the user through this window. These details are usually displayed in a read-only format, without any input from the user. In this way, the user gets a quick overview of the machine and does not encounter a large amount of data on the UI.

Figure 3.6: Machine Details modal

The second modal opens when the ”SEE MESSAGES” button is clicked and contains a detailed list of messages related to the machine. These messages; It contains fields such as message ID, machine ID, content (`Content`), read status (`IsRead`) and message creation time (`CreatedAt`). Data is dynamically filled in from the server and presented to the user in a tabular format. If desired, the user can select messages individually or collectively and mark them as read with the ”MARK AS READ” button. In this way, the user not only monitors the production line, but also interacts with the history of the system. Displaying unread messages prevents missing critical production notifications.[26]

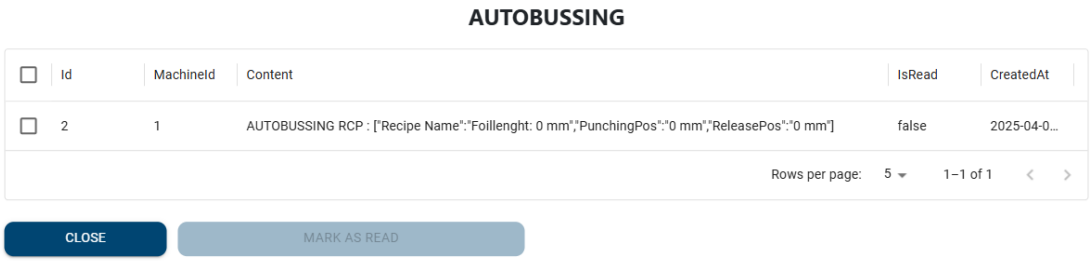

Figure 3.7: Machine Messages Modal

In addition, this entire structure can be updated in real time with new messages coming from the backend via SignalR; in other words, the user can see new messages without refreshing the page. This transforms the system from a monitoring panel to an active and responsive monitoring solution.

In the Supervisor application, the management of machine data is performed via the Redux structure on the client side. When the user interface is opened or a certain action is performed, the system sends a GET request to the `/Machine` API endpoint on the server with an asynchronous Redux action called `getAllMachines`. The response received as a result of this request is transferred to the Redux store, which provides state management for the application. The properties of each machine in the response are dynamically parsed, and from these, the header information (e.g. column names) used by components such as tables in the interface are also derived. Then, this data is successfully processed into the Redux store as a structured object.

The process of writing data to the store is handled in the reducer layer. If the data coming from the server represents more than one machine, the reducer writes this data directly to the store. However, if the aim is to update only a single machine, only the record with the matching id value in the machine list is updated thanks to the map function in the reducer, while the others are kept as they are. This structure ensures that the application works in synchrony with the user interface and allows the updated data to be effective only in the relevant area. With Redux, the user can view changes in data without refreshing the page. This approach provides high efficiency and stability in real-time production monitoring applications.

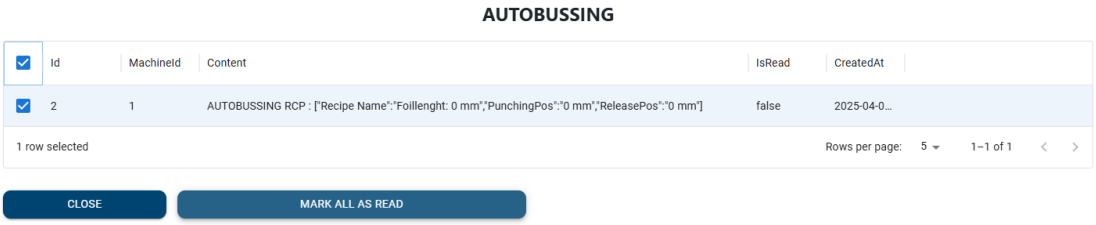

Figure 3.8: Messages can be marked as read

As a conclusion, these two modal components are complementary modules that deepen the user interaction initiated with `MachineCard` and allow more information exchange with the system. It both facilitates access to information and actively includes the user in the decision-making processes of the system.

4

Execution and Results

4.1 Running the Supervisor in Local Environment

This section explains in detail how the developed software is run and how it is installed in the test environment. The Supervisor project is a client-server based system with a multi-layered architecture. It consists of a React-based user interface (front-end), a .NET-based Web API (back-end), an MQTT listener service running in the background, and database layers. Certain steps must be followed to ensure that this structure is properly implemented in a local development environment.

4.1.1 System Requirements

In order for the project to run smoothly in a local environment, the following software must be installed on the system:

- Visual Studio 2022 or later (required workload: .NET desktop & ASP.NET development)
- .NET 8.0 SDK
- Node.js (v18 and later, for React)
- npm or yarn (for package management)
- Microsoft SQL Server (Express or Developer version is sufficient)
- SQL Server Management Studio (SSMS) (for database monitoring and table creating)

- MQTT Broker (tests have been done with MQTT Explorer)
- Postman or Swagger (for API testing, in this testing swagger have been used)

4.1.2 Database and Entity Framework Configuration

In the Supervisor project, data access is performed via Entity Framework Core. The `SupervisorDbContext.cs` class defined in the Supervisor.Data project is the structure that manages all DbSets and database connections. The database connection address is defined in the `appsettings.JSON` file in the Supervisor.Server project under the ”ConnectionStrings” section:

```
//..
"ConnectionStrings": {
  "connectionString": "Data Source=(localdb)\\MSSQLLocalDB;Initial
   Catalog=Supervisor;Integrated Security=SSPI;"
}
```

Code 4.1: Connection string

Then tables are created via MSSQL:

```
USE [Supervisor]
GO

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Machines](
  [Id] [bigint] IDENTITY(1,1) NOT NULL,
  [Name] [nvarchar](max) NULL,
  [Status] [nvarchar](max) NULL
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
```

Code 4.2: Machine Table

```
CREATE TABLE [dbo].[Messages](
  [Id] [bigint] IDENTITY(1,1) NOT NULL,
```

```
    [MachineId] [bigint] NULL,
    [Content] [nvarchar](max) NULL,
    [IsRead] [bit] NULL,
    [CreatedAt] [datetime2](7) NOT NULL
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
```

Code 4.3: Message Table

4.1.3 Starting the Backend (.NET Web API) Services

In the server side of the Supervisor application, all services are started, dependencies are loaded, middleware layers are defined, and the application communicates with the outside world is done in the Program.cs file. This file is the starting point of the application's lifecycle and is the unit where configurations are defined centrally.

When the application is started, the components that need to be added to the service collection are first defined. In this context, controller services that enable the processing of HTTP requests are added, and the SignalR infrastructure used for real-time communication is added. In addition, Swagger/OpenAPI support is provided, allowing to test API endpoints interactively.

The database connection is configured and the application connects to SQL Server via the connectionString parameter defined in the `appsettings.JSON` file. In this context, all data operations are performed with the Entity Framework Core ORM layer.

The repository and service classes defined as a requirement of the layered architecture in the application have been added to the container with the Dependency Injection principle. For example, the `IMachineRepository` interface is directed to the `MachineRepository` class when the application runs. Similarly, services such as `IMachineService` and `IMessageService` have been injected into the DI container in a way that corresponds to their own interfaces. This structure makes the dependencies of the application controllable and testable.

The session management of the application is also structured here. User-specific sessions are defined and the duration of these sessions is set to 60 min-

utes. In addition session data can be stored in memory using specific session functions.

In terms of security, CORS (Cross-Origin Resource Sharing) settings are defined comprehensively. With the AllowAnyHeader, AllowAnyMethod and AllowCredentials options, client applications from different origins (e.g. React frontend) can access the API. Allowing such flexible settings during the development phase ensures that test processes are carried out more efficiently, especially in the `localhost` environment. Allowing any header, method and credentials while defining the CORS policy is dangerous in real-life applications. Allowing any credential and headers in the CORS policy has been shown to create a CORS vulnerability that allows sensitive data to be stolen. Misconfigurations are known to allow attackers to perform denial-of-service (DoS) and steal data. In supervisor this policy only used in local development and for testing purposes.[27]

Finally, thanks to the `MapFallbackToFile("/index.html")` function, since the front-end side is SPA (Single Page Application), unknown routes are automatically directed to the `index.html` file. This is an important configuration for the correct rendering of the React application.

Because of this structure, the Supervisor server becomes a central platform that manages the database, messaging infrastructure, service layers and client communication. The Program.cs file is not only for configuration purposes, but also a strategic component that carries the backbone of the application architecture.

4.1.4 Running Front-end (React) Application

The client side is developed with React under the supervisor.client folder. This application provides the user interface and connects to the back-end with a REST API. The following steps should be followed to launch the application:

```
1 cd supervisor.client
2 npm install
3 npm run dev
```

Code 4.4: Running client-side

The following configurations must be correct in the .env file within the React application:

```
// These addresses can change according to environment.
VITE_API_URL=https://localhost:5001/api
VITE_SIGNALR_HUB_URL=https://localhost:5001/messageHub
```

Code 4.5: .env file configuration

With this information, the front-end application connects to the correct back-end and SignalR hub addresses. When the SignalR connection is successful, the user can see new messages instantly on the screen. User interactions are managed with the help of libraries such as Redux, axios and toastify.

When all layers are run locally, the Supervisor application allows the user to monitor machines, track messages, view historical records and communicate in real time. The local test environment allows the system to be tested safely and synchronously throughout the development process. This structure also forms the basis for moving the system to the server environment (IIS installation) in the later stages.

4.2 Publishing Supervisor Application via IIS

After the Supervisor project is developed locally, the application needs to be hosted on a web server so that it can reach a wider range of users. For this purpose, Microsoft's official web server, Internet Information Services (IIS), was used. In this section, how the .NET-based web API project Supervisor.Server is installed and configured on IIS is explained step by step.

4.2.1 Deployment Process of APIs

Before the installation on IIS, the ASP.NET Core project must be published as executables and content. This process can be done via Visual Studio:

- The Supervisor.Server project is selected in Visual Studio.
- Right-click and select the "Publish" option.
- A new publish profile is created as Folder Profile.

- The path C:/inetpub/Supervisor.Server is selected as the target folder.

- The Release build profile (build configuration) is selected.

- The publishing process is completed.

As a result of this process, the compiled .dll, web.config, appsettings.JSON and other necessary files are found in the relevant folder. This process must be done for all APIs.

4.2.2 Deployment of Supervisor.Client (React Application)

Supervisor.Client is a single page web application (SPA) developed with React. This application, which runs with the npm run dev command in the local development environment, must first be converted into static files in order to be transferred to the production environment.

The React project is compiled from the terminal with the following commands:

```
cd supervisor.client
npm install
npm run build
```

Code 4.6: .env file configuration

After this command, a folder named dist or build is created. This folder contains all the HTML, JS and CSS files of the application. These files are static and can be hosted on any server. The contents of the dist or build folder that is created are copied to the Supervisor.Server/wwwroot folder.

4.3 Execution

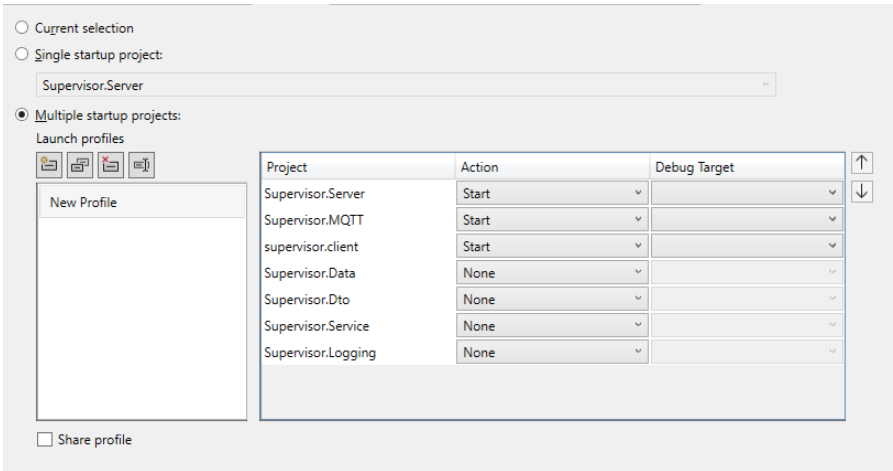

Figure 4.1: Configuring startup projects

Only necessary projects; `Supervisor.MQTT`, `Supervisor.Server` and `Supervisor.client` have been configured as 'Start'. This makes it easy for executing Supervisor application and consumes less memory.

Now, just clicking start and the application will be running.

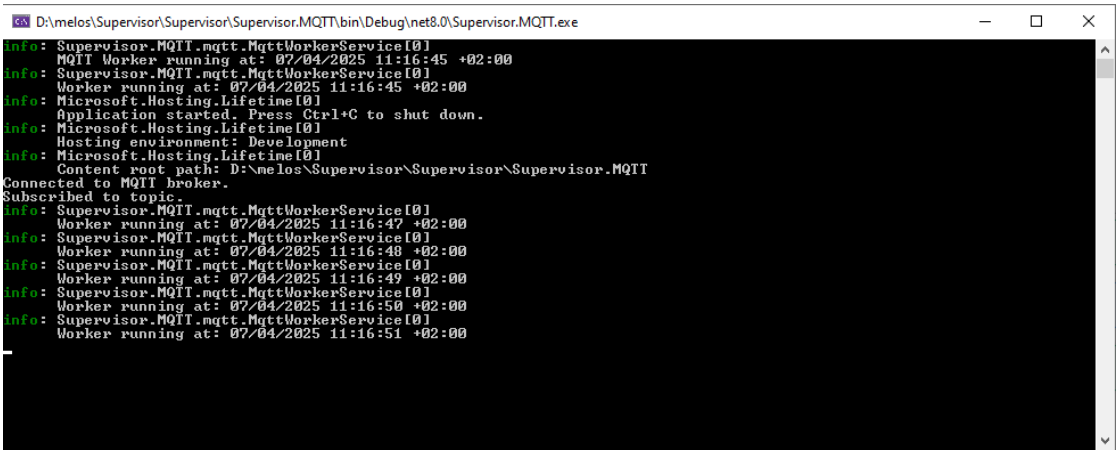

Figure 4.2: MQTT connection established

`Supervisor.MQTT` is up and running; waiting for messages from MQTT broker.

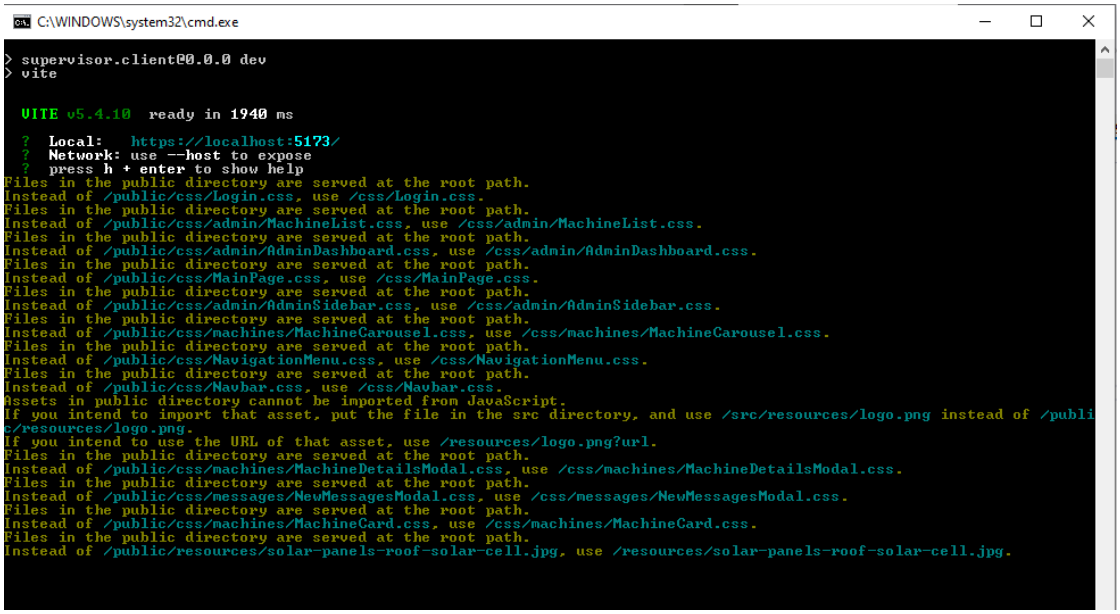

Figure 4.3: Client side running

`Supervisor.React` is successfully running.

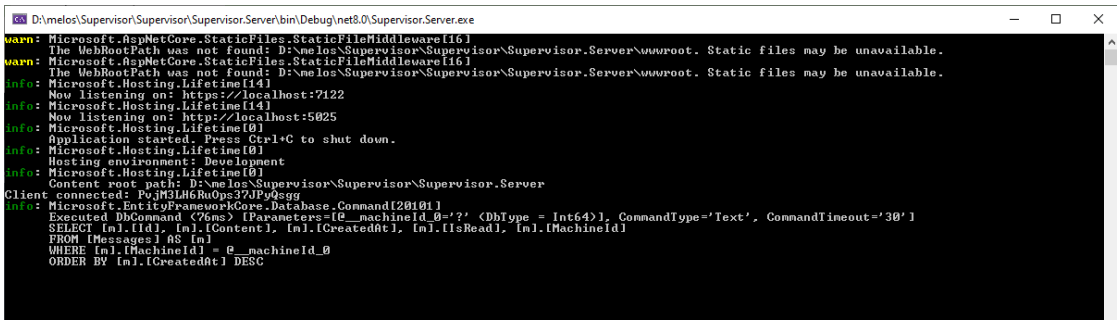

Figure 4.4: Server side executed

Lastly, `Supervisor.Server` is successfully executed and ready to give responses to client side.

MQTT Explorer

MQTT Explorer is a powerful tool that allows you to follow messages between clients communicating using the MQTT protocol through a visual interface. Thanks to this application, the behavior of both publishers and subscribers can be observed, and message content can be analyzed in detail.

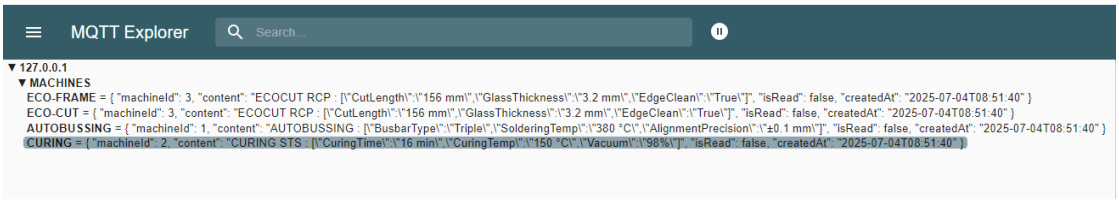

Figure 4.5: Messages tracked from MQTT Explorer

As seen in the figure above, different machines in the system (such as AUTOBUSSING, CURING, ECO-CUT and ECO-FRAME) send messages via MQTT with specific topics. Each message is transmitted in the form of a JSON object and includes fields such as machine ID, message content, read status and creation date.

Since the MQTT Explorer tool listens to such messages directly and displays them in a structured manner, it provides great convenience in checking the system's communication accuracy, data structure and up-to-dateness. In this way, software development and system testing processes can be carried out faster, more reliably and more visually.

Swaggers

In the Supervisor application, the Swagger interface has been activated and all endpoints (e.g. api/Machine, api/Message, api/Authentication) have been visually listed. Through this interface, information such as the request type (GET, POST, PUT, DELETE) of each API method, the URL from which it will be accessed, the parameters it should receive, and the type of response it returns can be examined in detail. During the development process, this structure has been used to test whether the services are working correctly, and the data formats expected from the clients have been easily verified. See example below:

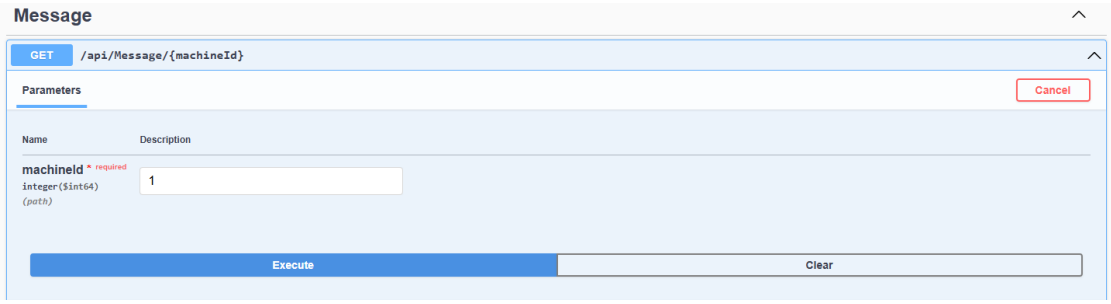

Figure 4.6: Swagger testing endpoint

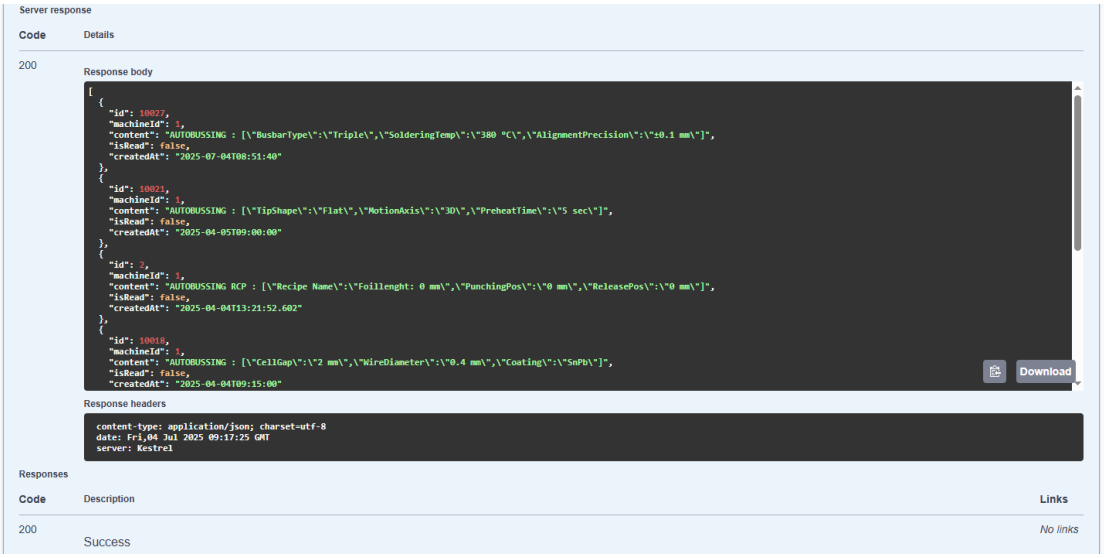

Figure 4.7: Swagger endpoint response

On the client side web page is opened and SignalR is connected, listening server side for any messages.

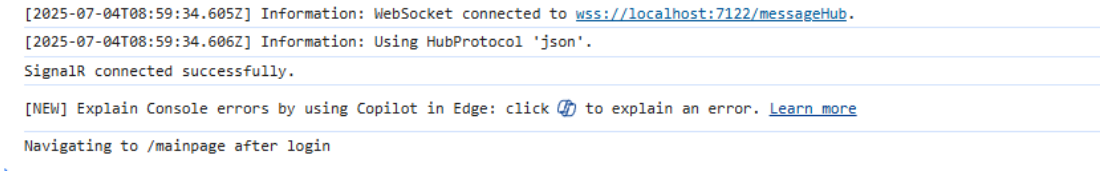

Figure 4.8: SignalR connection can be seen from developer tools(F12)

Messages coming from machines are captured with Supervisor.MQTT in the background of the system and then processed. These messages are both saved in the database and instantly transmitted to all connected clients via SignalR technology. On the client side, application listens for the SignalR connection and presents the incoming messages to the user as a visual notification. With this, the data flow in the system is completed in real time from end to end.

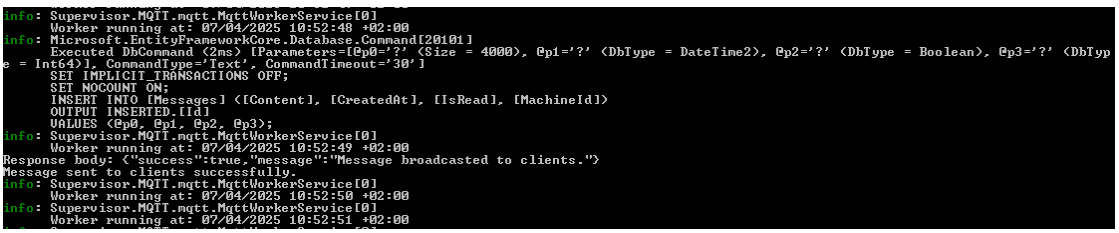

Figure 4.9: Supervisor.MQTT catching messages

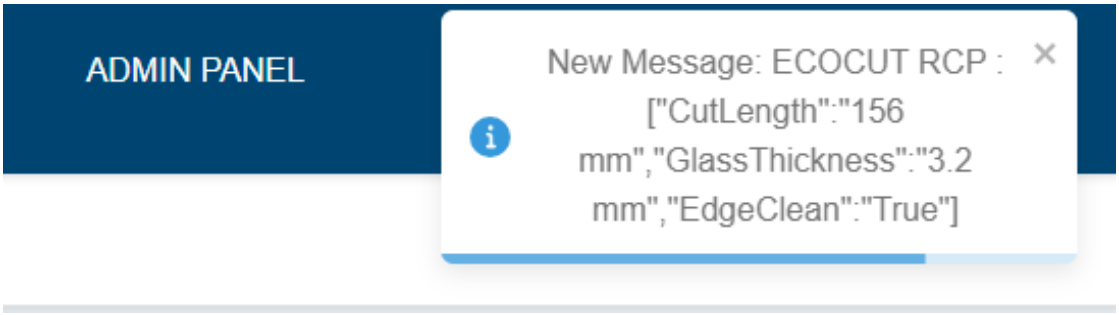

Figure 4.10: Instant message on the application

With this logic, the application continues to insert messages into the database and shows them to the user.

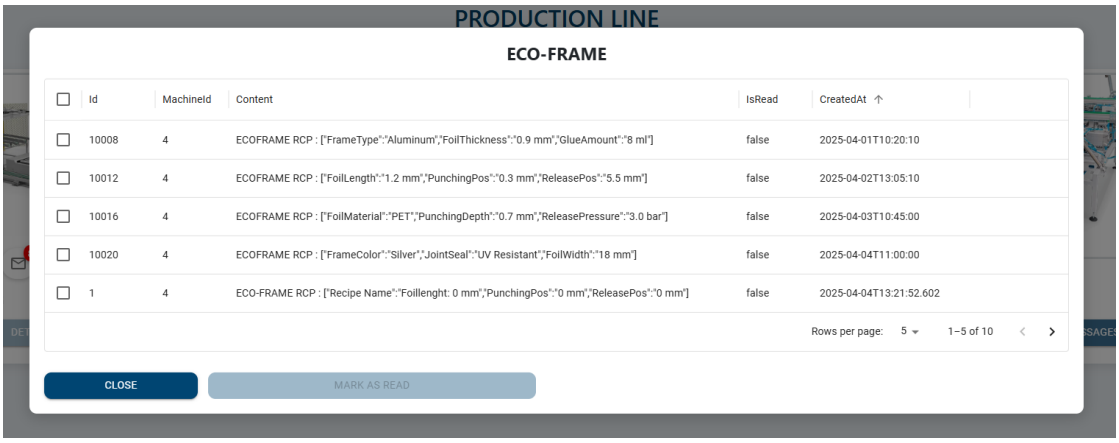

Figure 4.11: Received messages of the machine

Performance Evaluation

To assess the system's performance under different message count, a controlled stress test was conducted using Python scripts to send MQTT messages to the Supervisor.MQTT backend. Four different message volumes—100, 200, 500, and 1000 messages—were tested. For each case, the total time taken to send all messages from the client side (Python script) and the time taken by the server to receive and process those messages were recorded. Below are the results of this tests.

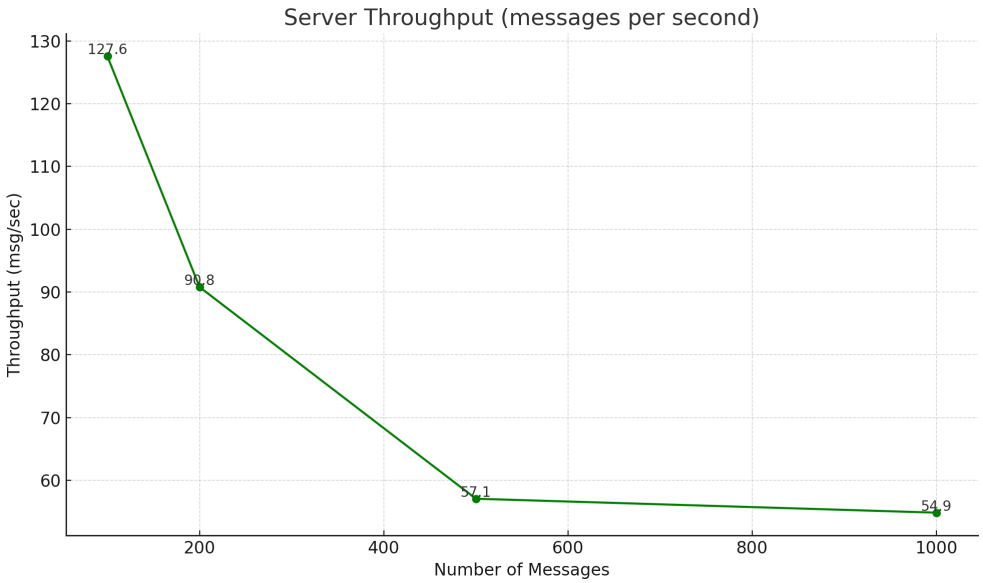

Figure 4.12: Server throughput graph

Messages	Python Sent (s)	Server Received (s)	Throughput (msg/s)
100	1.68	0.78	127.6
200	3.73	2.20	90.9
500	10.33	8.76	57.0
1000	19.77	18.22	54.9

Figure 4.13: Measured Performance Metrics During Stress Test

In order to better understand the system's behavior under increasing load, a more in-depth analysis was carried out by comparing three key metrics: the total message sending time from the client (Python script), the total message processing time on the server (Supervisor.MQTT), and the effective throughput measured as the number of messages received per second by the server. These results are visualized in Figure 4.12 and summarized numerically in Figure 4.13.

From the chart and table, it can be observed that as the number of messages increased both the Python sending time and the server processing time increased in near-linear trend as visualized in Figure 4.14 below. However, the sending time was consistently higher than the server's receiving time. For instance, in the 1000-message test, Python completed sending in 19.77 seconds, whereas the

server received and processed the same messages in 18.22 seconds. This behavior was consistent across all test volumes.

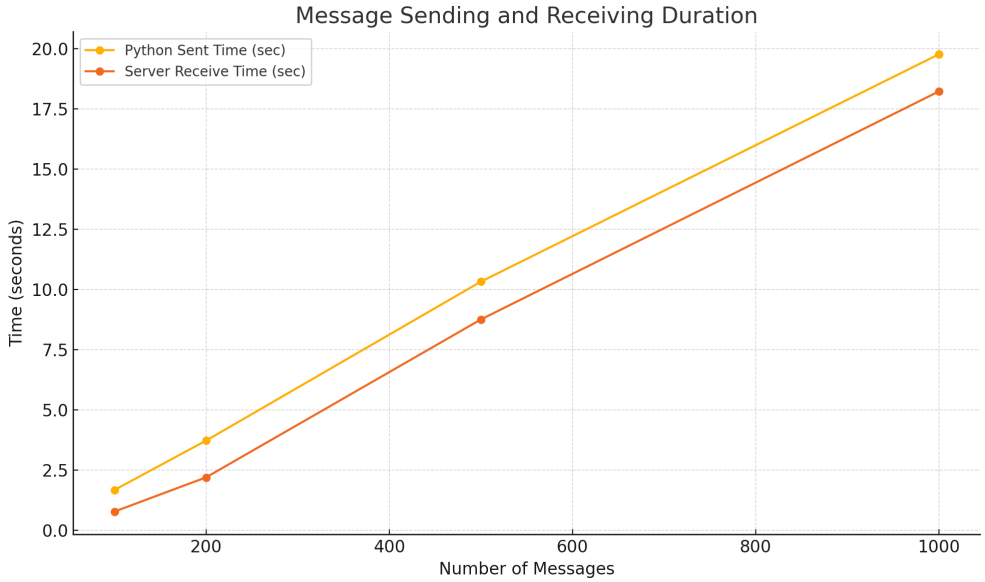

Figure 4.14: Message duration graph

This difference is expected because: In python script paho.mqtt have been used which performs synchronous publish() operations sequentially and includes a slight delay (sleep(0.01)) between messages to avoid flooding the broker.

In contrast, the server (Supervisor.MQTT) is set up to handle messages asynchronously and in parallel, using efficient background processing (via BackgroundService) and a lightweight deserialization pipeline. Under production-grade conditions sending time and receiving time would converge, and the throughput would become more stable.

Another result from this experiment is the gradual decrease in throughput as the message volume increases. The system was able to handle 100 messages at a throughput of 127.6 msg/sec, but this number decreased to 54.9 msg/sec for 1000 messages. This is typical and reflects the expected behavior of a single-channel message publishing strategy.

5 Conclusions and Future Works

5.1 Conclusion

Within the scope of this thesis, the design and implementation of a modular, real-time monitoring and communication system for solar panel production lines has been carried out. The system was developed specifically targeting Ecoprogetti's production environment; MQTT protocol was used for message transfer and SignalR for real-time notifications. Using a multi-layered architecture that combines technologies such as ASP.NET Core, React and SQL Server, data coming from the machines was processed, stored and presented to the user in a fast and reliable manner.

One of the most important outputs of this project is the implementation of a service architecture that listens to and manages messages coming via MQTT in the background. Incoming messages are parsed and saved to the database using specific handlers, then transmitted to the client side instantly using SignalR. The front end of the application was developed with React and Redux technologies and offers an intuitive interface where users can monitor the current status of the machines, access messages and interact with the system.

In general, the developed software system has managed to provide a robust, expandable and user-friendly solution for the effective monitoring and management of production line machines. The system; by combining machine-level data flow with human-centric monitoring processes, it has increased operational

visibility and rapid response capacity.

5.2 Future Works

Although the current system provides a solid and functional foundation, there are many areas for future development. First, the MQTT configuration is currently limited to static topics and fixed credentials. Enhancing this structure with a dynamic subscription model and secure identity management (e.g. via Azure Key Vault or environment variables) will be beneficial in terms of security and flexibility.[28]

Another area of development is adding an analysis and reporting module to the system. By storing and analyzing past messages; data such as machine availability, error rates, downtimes can be visualized and the system can provide basic predictive maintenance suggestions. Such visualizations will make it easier for production managers to make data-driven decisions.[29]

All MQTT messages are currently processed in the system in the same way. However, different workflows may be required depending on the message type. Therefore, creating a message routing structure where incoming messages are classified according to their type or topic hierarchy and directed with special business logic will increase the extensibility of the system.[30]

In addition, in the current version of the system, the communication structure is only one-way; messages from machines are forwarded to the central server. However, in the future, it is planned to make the system suitable for two-way communication. In this direction, operators can be provided with the ability to send certain commands (such as “stop the machine”, “reset the alarm”, “update recipe”) directly to the machines via the Supervisor interface via the MQTT protocol. Such a structure enables not only monitoring but also remote control features and transforms the system into a full-fledged SCADA-like platform.

On the client side, more features can be added to the admin panel. Features such as user role management, logging, and exporting past messages can be integrated into this section. In addition, a mobile application developed with

React Native for the accessibility of the system from mobile devices can bring the real-time monitoring experience to the field.

Finally, in order to increase the sustainability of the system, it is of great importance to establish a comprehensive test infrastructure by writing unit tests, integration tests, and system-level tests.

References

[1] John R. Balfour, Michael Shaw, and Nicole Bremer Nash. *Introduction to Photovoltaic System Design*. Jones & Bartlett Publishers, 2013.

[2] Feyza Akarslan. "Photovoltaic systems and applications". In: *Modeling and Optimization of Renewable Energy Systems* 3 (2012), pp. 1035–1041.

[3] M.O. Karaağaç, A. Ergün, O. Arslan, and M. Kayfeci. "Introduction to solar panels". In: *Handbook of Thermal Management Systems*. 2023, pp. 541–556.

[4] Firas Obeidat. "A comprehensive review of future photovoltaic systems". In: *Solar Energy* 163 (2018), pp. 545–551.

[5] Greta Vallero, Ana Cabrera-Tobar, Giovanni Perin, Daniela Renga, Leonardo Badia, Michele Rossi, Michela Meo, Francesco Grimaccia, and Sonia Leva. "Network Resilience and Sustainability: Renewable Energy-Based Solutions". In: *IEEE Communications Magazine* 63.5 (2025), pp. 118–124.

[6] Artiom Blinovas, Kenji Urazaki Junior, Leonardo Badia, and Elvina Gindullina. "A game theoretic approach for cost-effective management of energy harvesting smart grids". In: *Proc. International Wireless Communications and Mobile Computing (IWCMC), IEEE*. 2022, pp. 18–23.

[7] Ayşegül Taşçıoğlu, Onur Taşkın, and Ali Vardar. "A power case study for monocrystalline and polycrystalline solar panels in Bursa City, Turkey". In: *International Journal of Renewable Energy Research* (2016).

[8] Turlough F. Guerin. "Evaluating treatment pathways for managing packaging materials from construction of a solar photovoltaic power station". In: *Waste Management and Research* 38.12 (2020), pp. 1345–1357.

[9] Ecoprogetti. *How to Manufacture Photovoltaic Module*. n.d. URL: `https://ecoprogetti.com/how-to-manufacture-photovoltaic-module/`.

[10] William Bolton. *Programmable logic controllers*. Newnes, 2015.

[11] L. Badia and T. Marchioro. "Distributed and Timely Smart Microgrid Management Through Markov Games". In: *5th International Conference on Clean and Green Energy Engineering (CGEE), IEEE*. 2024, pp. 32–36.

[12] Elias Kai. *Happy Birthday MQTT*. Accessed: 2025-07-04. 2009. URL: `https://web.archive.org/web/20150315025826/https://mqtt.org/2009/07/10th-birthday-party`.

[13] Valéry Masson, Marion Bonhomme, Jean-Luc Salagnac, Xavier Briottet, and Aude Lemonsu. "Solar panels reduce both global warming and urban heat island". In: *Frontiers in Environmental Science* 2 (2024), p. 14.

[14] Farag Azzedin and Turki Alhazmi. "Secure data distribution architecture in IoT using MQTT". In: *Applied Sciences* 13.4 (2025), p. 2515.

[15] Leonardo Badia and Andrea Munari. "Exogenous Update Scheduling in the Industrial Internet of Things for Minimal Age of Information". In: *IEEE Transactions on Industrial Informatics* 21 (2025), pp. 1210–1219.

[16] Se-Chun Oh and Young-Gon Kim. "A Study on MQTT based on Priority Topic for IIoT". In: *The Journal of the Institute of Internet, Broadcasting and Communication* 19.5 (2019), pp. 63–71.

[17] Leonardo Badia. "Analysis of age of information under SR ARQ". In: *IEEE Communications Letters* 27.9 (2023), pp. 2308–2312.

[18] J. Singh and D. S. Kapoor. "QoS-based Performance Evaluation of MQTT Protocol for IoT Applications". In: *Journal of Internet of Things and Applications* (2021), pp. 12–14.

[19] S. Pradhan and R. A. Khan. "Enhancing Fault Detection in MQTT Networks using Last Will and Testament Feature". In: *Proc. International Conference on IoT Systems*. 2022, pp. 8–10.

[20] H. Wang and J. Li. "Design and Implementation of MQTT-based Sensor Network with Retained Messages for Indoor Monitoring". In: *International Conference on Indoor Sensing Systems*. 2019, pp. 5–6.

[21] N. Edwin. "Software Frameworks, Architectural and Design Patterns". In: *Journal of Software Engineering and Applications* 7 (2024), pp. 670–678.

[22] Songtao Chen, Upendar Rao Thaduri, and Venkata Koteswara Rao Ballamudi. *Front-End Development in React: An Overview*. Unpublished manuscript or technical report. 2023.

[23] Johan Franz and Milton Niklasson. “A learning curve comparison between React and Angular”. Bachelor’s thesis. Chalmers University of Technology, 2024.

[24] Krutika Patil. “Redux State Management System – A Comprehensive Review”. In: *International Journal of Trend in Scientific Research and Development* (2022), pp. 1022–1026.

[25] Leonardo Badia and Andrea Munari. “Status update scheduling in remote sensing under variable activation delay”. In: *IEEE International Balkan Conference on Communications and Networking (BalkanCom)*. 2023, pp. 1–6.

[26] Alessandro Buratto, Begüm Yivli, and Leonardo Badia. “Machine learning misclassification within status update optimization”. In: *IEEE International Conference on Communication, Networks and Satellite (COMNETSAT)*. 2023, pp. 640–645.

[27] Matteo Golinelli, Elham Arshad, and Bruno Crispo. “Mind the CORS”. In: *IEEE 5th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications*. 2023, pp. 213–221.

[28] Jochen Nickel. *Mastering Identity and Access Management with Microsoft Azure: Empower users by managing and protecting identities and data*. Packt Publishing Ltd, 2019.

[29] Alessandro Bertoni, Xin Yi, Claude Baron, Phillippe Esteban, and Rob Vingerhoeds. “A framework for data-driven design in a product innovation process: data analysis and visualisation for model-based decision making”. In: *International Journal of Product Development* 24.1 (2020), pp. 68–94.

[30] Iffat Ahmed, Leonardo Badia, and Khalid Hussain. “Evaluation of deficit round robin queue discipline for real-time traffic management in an RTP/RTCP environment”. In: *Proc. Fourth UKSim European Symposium on Computer Modeling and Simulation, IEEE*. 2010, pp. 484–489.

[31] Alberto Zancanaro, Giulia Cisotto, and Leonardo Badia. “Modeling value of information in remote sensing from correlated sources”. In: *Computer Communications* 203 (2023), pp. 289–297.

[32] Emilio J. Juarez-Perez, Rafael S. Sanchez, Laura Badia, Germá Garcia-Belmonte, Yong Soo Kang, Ivan Mora-Sero, and Juan Bisquer. “Photoinduced giant dielectric constant in lead halide perovskite solar cells”. In: *The Journal of Physical Chemistry Letters* 5.13 (2014), pp. 2390–2394.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my thesis advisor, Leonardo Badia, for their continuous guidance, insightful feedback, and academic support throughout the entire internship and development process. Their expertise and encouragement have been essential in shaping this work.

I am sincerely thankful to Ecoprogetti Srl, the company that hosted my internship and provided a meaningful environment where this project could take shape. Special thanks to my mentor Antonio Pegoraro, who generously shared their technical knowledge and offered valuable feedback at every stage of the implementation.

Beyond the academic and professional support, I owe a heartfelt thank you to my family and friends, whose patience, encouragement, and emotional strength have carried me through all the challenges of this thesis journey.

I would like to extend my deepest respect and gratitude to my beloved country, Türkiye, for giving me the opportunity to pursue higher education abroad and represent its values with pride.

Lastly, I would like to dedicate this work to Mustafa Kemal Atatürk, the founder of the Republic of Turkey, whose vision for education, science, and progress continues to inspire generations, including mine.