

**BABEȘ-BOLYAI UNIVERSITY CLUJ-NAPOCA**  
**FACULTY OF MATHEMATICS AND COMPUTER**  
**SCIENCE**  
**SPECIALIZATION COMPUTER SCIENCE**

## **DIPLOMA THESIS**

# **EDI Integration Manager for Interchange Message Processing**

**Supervisor**  
**Lect. Dr. Tudor-Dan Mihoc**

*Author*  
*Iaguta Alen-Mihael*

2025



---

## ABSTRACT

---

This thesis presents the design and implementation of an EDI (Electronic Data Interchange) Integration Manager web application, a full-stack solution made with .NET background and an Angular frontend, being powerful and easy to use. The application functions as an intuitive free platform for comprehensive managing and organization of EDI files, addressing business requirements for essential digital document exchange.

The project encompasses several key components: data storage, seamless reception and reliable transmission of EDI files. Each component is designed to ensure robustness, accuracy, and efficiency. In addition, the application offers advanced functionalities such as the creation and generation of new EDI content, customization of specific EDI content to individual companies, respecting diverse trading partner requirements and optimizing data exchange for unique business relationships.

Ultimately, this EDI Integration Manager is designed to streamline EDI workflows for modern businesses, in an easy and low cost manner. By providing an ecosystem for managing all EDI operations - from content creation to secure transmission and simple storage - the system empowers organizations to achieve high efficiency, minimize manual errors, and to overall automate their business processes. Future work, will focus on optimizing and expanding the message transmission protocols, boosting the user experience and exploring new research opportunities in the realm of EDI technology.

# Contents

<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
<b>2</b>	<b>UNDERSTANDING ELECTRONIC DATA INTERCHANGE (EDI)</b>	<b>4</b>
2.1	Introduction to EDI . . . . .	4
2.1.1	Definition & Purpose of EDI . . . . .	5
2.1.2	EDI Benefits & Importance in Modern Business . . . . .	5
2.2	EDI Transactions . . . . .	6
2.2.1	Diverse Message Transactions . . . . .	6
<b>3</b>	<b>CURRENT STATE OF EDI INTEGRATION</b>	<b>7</b>
3.1	SAP . . . . .	7
3.2	Axway TSIM . . . . .	7
<b>4</b>	<b>TECHNOLOGIES USED</b>	<b>9</b>
4.1	Integration with .NET Server . . . . .	9
4.2	Angular GUI . . . . .	11
<b>5</b>	<b>APPLICATION IMPLEMENTATION</b>	<b>12</b>
5.1	Specification . . . . .	12
5.2	Server Architecture . . . . .	13
5.2.1	Database Design and Models . . . . .	13
5.2.2	Repository Layer . . . . .	21
5.2.3	Controllers . . . . .	23
5.2.4	Services . . . . .	33
5.3	Web Interface Design . . . . .	36
5.3.1	Authentication . . . . .	37
5.3.2	API Communication . . . . .	38
5.3.3	Components Overview . . . . .	39
5.3.4	UI Logic . . . . .	40
5.4	Integration Flow . . . . .	42
5.4.1	Partner . . . . .	42

5.4.2	Workflow . . . . .	44
5.4.3	Rule . . . . .	45
5.4.4	Messages . . . . .	47
5.4.5	Message Generation . . . . .	48
5.4.6	Message Processing . . . . .	48
5.4.7	Archive . . . . .	49
<b>6</b>	<b>RESULTS &amp; PROSPECTS</b>	<b>51</b>
6.1	Future Work . . . . .	51
6.1.1	XML Conversion . . . . .	51
6.1.2	User-Defined Workflows . . . . .	52
6.2	Conclusions . . . . .	53
	<b>Bibliography</b>	<b>54</b>

# Chapter 1

## INTRODUCTION

In today's interconnected business environment, the seamless exchange of information is critical for the efficiency and effectiveness of operations. Electronic Data Interchange (EDI) has emerged as a vital technology that allows businesses to exchange documents electronically in a standardized format. EDI replaces traditional paper-based communication, significantly reducing the time, costs, and errors associated with manual processing. Despite its importance, the underlying processes and technology of EDI often go unnoticed, functioning quietly in the background to support critical business functions.

EDI is particularly crucial for industries such as retail, manufacturing, and logistics, where timely and accurate data exchange is essential. It enables automated transactions, including purchase orders, invoices, shipping notices, and other business documents [Thu20]. Companies relying on EDI benefit from faster transaction processing, improved accuracy, and enhanced supply chain visibility. As a result, many businesses find it nearly impossible to operate efficiently without EDI, making it a foundational technology in the modern business sphere.

This thesis aims to design and implement an EDI Integration Manager web application, having its primary goal to create a tool that can centralize and automate the management of business processes through EDI. Providing a versatile platform, supporting key standards like ANSI X12 and EDIFACT, the Integration Manager allows also data exchange, automated file reception and transmission, message translation, error handling, and real-time monitoring and reporting. Its customizable workflow engine allows businesses to define actions for incoming and outgoing EDI messages, integrating with internal business processes.

The primary contributions of this thesis include:

- **Development of an EDI Integration Manager:** The creation and evaluation of a full-stack web application that automates EDI operations.
- **Support for Diverse EDI Standards:** It presents a versatile platform capable of handling ANSI X12 and EDIFACT standards, addressing adaptability in modern EDI.
- **Practical Case Study and Strategic Insights:** The work provides a real-world case study of how a cost-effective EDI integration solution can enhance efficiency and accuracy in business-to-business transactions.

The thesis is organized into several chapters, each addressing a specific aspect of the project:

- **Introduction:** Provides the background, motivation, and objectives of the thesis.
- **Literature Review:** Reviews existing EDI technologies and related work.
- **Design and Implementation:** Details the design choices and architectural blueprint of the application.
- **Implementation of the EDI Integration Manager:** Explains the practical development of the system, covering also the user interface.
- **Business Process Management:** Demonstrates how the application manages and automates business processes through use cases.
- **Results and Discussion:** Presents the results of the implementation, including performance benchmarks and validation outcomes.
- **Conclusion and Future Work:** Summarizes the findings, discusses the significance of the work, and outlines potential future directions.

In summary, this thesis addresses the critical need for advanced EDI processing tools by developing a comprehensive integration manager. The project not only enhances the understanding of EDI technologies but also provides a practical tool for businesses to streamline their data exchange processes. The following chapters will dig into the details of the design, implementation, and evaluation of the system, providing a thorough examination of the project's contributions and future potential.



## **Chapter 2**

# **UNDERSTANDING ELECTRONIC DATA INTERCHANGE (EDI)**

The rapid expansion of information technology (IT) offers significant potential to enhance organizational performance. However, substantial investments in IT create pressure on management to justify these costs by demonstrating tangible business value. In the realm of transport services, IT has enabled the development of faster, more reliable, and precisely timed logistics strategies, making information-intensive transportation services central to modern operations. The internationalization of production networks and the competitive emphasis on time efficiency have intensified the importance of efficient supply chain operations, according to [Jan11].

## **2.1 Introduction to EDI**

Synchronization issues in supply chains can lead to immobilized goods, resulting in storage, security risks, and investment costs. This highlights the urgent need for aligning document flows with goods movement through automation and electronic data interchange (EDI), as well as standardizing and simplifying trade procedures and documents. EDI and other inter-organizational systems, such as e-mail, facilitate 'vertical information integration' between trading partners, enhancing the accuracy and timeliness of information exchange.

With the rise of global business and intermodal transportation, maritime transport has gained importance, with ports serving as critical hubs. The competition among ports has intensified, and information technology, particularly EDI, has become essential for strengthening port operations. EDI's high speed, reliability, and ease of data capture make it a preferred tool for information exchange in port communities.

The strategic advantages associated with EDI are contingent upon the incorporation of information obtained from external sources into the current organizational systems and practices; as well as the integration of those systems and practices themselves - a process that has the potential to transform the entire framework of the organization in question. [SS92]

### **2.1.1 Definition & Purpose of EDI**

EDI is defined in a remarkably similar manner by the vast majority of writers on the subject, as the following sample of the many available definitions indicates:

- The standards-based computer-to-computer exchange of intercompany business documents and information" [Coa88]
- Computer-to-computer transmission of standard business data" [Emm90]
- The computer-to-computer exchange of standard business documents between trading partners. This includes the company's suppliers, its customers and its bank" [SSF94]
- The movement of business data electronically between or within firms in a structured, computer processable data format that permits data to be transferred without rekeying from a computer supported business application in one location to a computer supported business application in another location [HF89]
- The replacement of the paper documents used in Administration, Commerce and Transport by electronic messages conveyed from one computer to another without the need for human intervention" [GSWI95]

### **2.1.2 EDI Benefits & Importance in Modern Business**

The exchange of information is crucial for freight movement, just as essential as the transportation of the cargo itself or the equipment facilitating that transport. In the realm of freight transportation, the absence of information flow directly impacts the movement of cargo. A more seamless flow of information enables faster transit of cargo from its point of origin to its final destination. Electronic Data Interchange (EDI) communications play a vital role in ensuring the efficient handling of cargo across different modes of transport, while also automating processes such as billing, data entry, tracking functions, and various information exchanges, including cargo manifests, vessel arrival schedules, inbound movements, and status updates. In this

context, EDI can significantly reduce cycle times, expedite document forwarding, enhance inventory management, optimize scheduling, and facilitate purchasing, all in an electronic and automated manner.

## 2.2 EDI Transactions

One of the most vital elements of EDI is that the information needs to be transmitted in a format that can be readily comprehended by a trading partner, irrespective of variations in computer systems. Since communication in EDI occurs between computers rather than between individuals, the data must be shared in a standardized format. This implies that the information should be organized in a predetermined, consistent structure that can be interpreted and understood by a computer without the need for human involvement.

### 2.2.1 Diverse Message Transactions

The two major factors that have been identified to contribute most to the rapid rate of EDI adoption, include external pressure to adopt EDI and the perceived benefits from using EDI [IBD95]. In addition to these driving forces, a third element contributes to the advancement of EDI, which is the overall improvement of individual EDI components. This encompasses technological advancements in both hardware and software, along with the organization and implementation of standards. The standardized electronic versions of these business or technical information forms that trading partners share through EDI are referred to as transaction sets. The most commonly transmitted types of transaction sets via EDI and their corresponding usage rates are as follows:

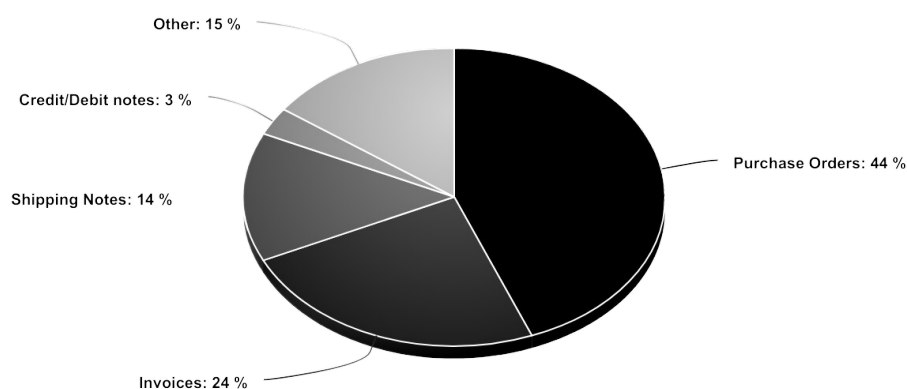


Figure 2.1: Types of EDI transactions

# Chapter 3

## CURRENT STATE OF EDI INTEGRATION

### 3.1 SAP

SAP (Systems, Applications, and Products in Data Processing) is a leading enterprise resource planning (ERP) software that integrates various business functions like finance, human resources, manufacturing, supply chain, and customer relationship management into a single, comprehensive system. SAP's goal is to unify data and business processes across an organization.

In the context of integration, SAP offers its own set of tools and platforms to facilitate data exchange both internally and with external partners. When SAP is integrated with EDI, the two systems work in tandem, automating the flow of information between trading partners. This integration reduces the need for manual data entry, ensures accuracy, and speeds up the exchange of critical business documents [She23].

it's crucial to recognize that while SAP provides the internal backbone for managing core business processes and has capabilities for handling EDI (often through IDocs), it isn't primarily designed as a dedicated B2B communication gateway.

### 3.2 Axway TSIM

Axway B2B Integration (including TSIM - Trade Settlement and Integration Manager) is a comprehensive platform designed to facilitate seamless business-to-business (B2B) interactions, data exchange, and integration across diverse systems and partners.

While SAP's only form of processing is internal (via IDocs), Axway TSIM provides a gateway to handle diverse B2B communication protocols and partner relationships, acting as a middleware layer between a SAP system and the trading network.

Feature	Axway TSIM	SAP
<b>Primary Role</b>	Dedicated <b>B2B Gateway &amp; EDI Orchestration</b>	Comprehensive <b>ERP &amp; Business Process Management</b>
<b>Scope</b>	External B2B communication; "outside-in" focus on connecting with trading partners	Internal business operations; "inside-out" focus on managing core enterprise functions
<b>EDI Handling</b>	Specialized platform for all EDI protocols, transformations, monitoring across diverse external partners	ERP has native EDI ( <b>IDoc</b> ) capabilities for internal processing; often integrates with external EDI solutions like TSIM for broader B2B connectivity
<b>Integration</b>	Integrates <i>with</i> ERPs (like SAP) and other internal systems to exchange B2B data	Integrates <i>across</i> its own modules (e.g., Finance, HR, Supply Chain) and can integrate with external systems/gateways
<b>Visibility</b>	Focus on the <b>B2B transaction lifecycle</b> visibility, tracking external message exchange status	Focus on overall business process and data visibility within the ERP system
<b>Vendor Type</b>	Integration and B2B specialist	Leading provider of ERP software and a wide range of business applications

# Chapter 4

## TECHNOLOGIES USED

In this chapter I will showcase the technologies used in my project, together with their purpose and benefits over other full-stack technologies.

### 4.1 Integration with .NET Server

.NET's main important selling point is its versatility and productivity for building high-performance, cross-platform applications, having one of the fastest application runtimes, thanks to its JIT (Just-In-Time) and AOT(Ahead-Of-Time) compilation. Because the JIT compiler only performs the first time a method is invoked, the methods you don't need at runtime will never be JIT-compiled. [TL03] Hence I choose .NET for the following reasons:

- **Entity Framework Core** offers a robust ORM(Object-Relational Mapping) which is very extensible - working across multiple database providers, in my case PostgreSQL, and also ensuring clean architecture with tight dependency injection.
- **Token-based authorization** is deeply integrated with ASP.NET Core and JWT middleware, making it one of the easiest-to-use services, when it comes to the authorization of endpoints. Having claims per model objects, this ensures the connectivity between each token and endpoint.
- **Database Migrations** are intuitive and version controlled, meaning that a schema evolution is present and easy to access at any step of evolution, abstracting complex SQL logic, allowing for no manually typed SQL code. I have showcased below, an example where we create a connection from table Messages to table User via UserId as foreign key having the purpose to assign messages to an user. This was generated automatically using .NET EF Core Migrations, which are applied then to the database:

```

1  public partial class UserMessageMigrations : Migration
2  {
3      /// <inheritdoc />
4      protected override void Up(MigrationBuilder
5          ↪ migrationBuilder)
6      {
7          migrationBuilder.AddColumn<Guid>(
8              name: "UserId",
9              table: "Messages",
10             type: "uuid",
11             nullable: false,
12             defaultValue: new Guid("
13                 ↪ 00000000-0000-0000-0000-000000000000"));
14
15         migrationBuilder.AddForeignKey(
16             name: "FK_Messages_Users_UserId",
17             table: "Messages",
18             column: "UserId",
19             principalTable: "Users",
20             principalColumn: "Id",
21             onDelete: ReferentialAction.Cascade);
22
23         migrationBuilder.CreateIndex(
24             name: "IX_Messages_UserId",
25             table: "Messages",
26             column: "UserId");
27     }
28
29     /// <inheritdoc />
30     protected override void Down(MigrationBuilder
31         ↪ migrationBuilder)
32     {
33         migrationBuilder.DropForeignKey(
34             name: "FK_Messages_Users_UserId",
35             table: "Messages");
36     }
37 }

```

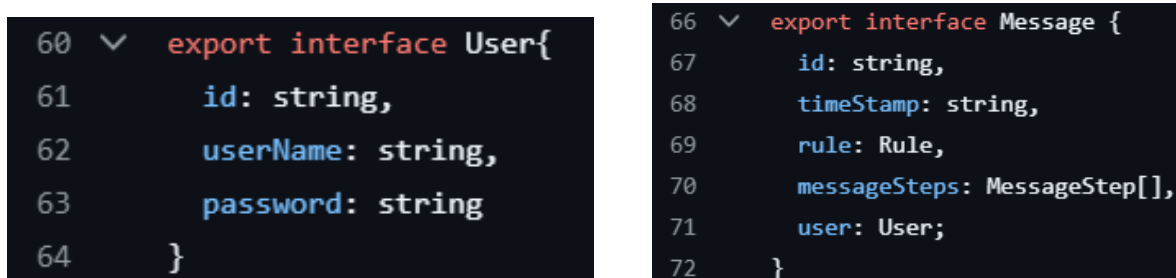
Listing 4.1: Adding a foreign key non-nullable UserId constraint to the Messages table.

- **Built-in Swagger(Swashbuckle)** provides support for generating OpenAPI documentation and enhancing API discoverability, having an easy to use starting page where the endpoints can be tested, hence I did not use external tools like Postman in order to test the application's endpoints.
- **Docker support** is coupled very tightly with .NET CLI and SDK, it allows containerization, environment consistency, and simplified deployment workflows.

## 4.2 Angular GUI

Angular is one of the most popular Javascript frameworks out there due to its ability to structure components in a hierarchical manner by design, its extensive set of tools that provide easy implementation for common use cases (e.g., routing, http calls) and its ease of use in both small and large scale projects.

I preferred Angular because it is a full-featured framework and not just a library. It includes packages which consists of Routing, Forms(both template-driven and reactive), HTTP client, state management and dependency injection, not having the need to use third-party libraries, all of this being integrated and maintained by Angular. It is different in that the template and data get shipped to the browser to be assembled there. [GS13]. Angular is also based on TypeScript, where TypeScript interfaces allow you to declare custom types. Interfaces prevent compile-time errors caused by using objects of the wrong type in an application.[MF18]. Below is the example of the migration shown in Listing 4.1.



```
60  export interface User{
61      id: string,
62      userName: string,
63      password: string
64  }

66  export interface Message {
67      id: string,
68      timeStamp: string,
69      rule: Rule,
70      messageSteps: MessageStep[],
71      user: User;
72  }
```

Figure 4.1: Custom types defined using TypeScript interfaces



# Chapter 5

## APPLICATION IMPLEMENTATION

The following section walk trough the process of implementing the Integration Manager application from the beginning. This includes an in-depth overview of the server structure, service logic, database design, and the development of the interactive UI interface.

### 5.1 Specification

This project is designed for start-up to small businesses in mind. The primary goal is to provide these companies with a practical environment for managing essential components like logistics or supply chain workflows. In addition, the application enables users to engage with real-world processes such as shipping material tracking, invoice management and the organization of associated documentation.

An angular-based interface and a server project written in C# using ASP.NET will be used to split the application. This allows for the development of numerous clients. Additionally, if the backend is running on an advanced computer, the UI code can run on a slower device.

All of the specific comments made in the previous chapter will be applied to the corresponding stack, as a result, the user interface will become more complex due to it's high number of models and actions to each model. Files are the foundation of this application - central to it's functionality and data structure. Each type of document can be linked to specific user-defined workflows and triggers within the system, allowing for actions such as validation, processing, error tracking, and archiving. This design, which is relied on documents, showcases real-world business operations, where access to these files and the management of these files is essential in maintaining efficiency in day-to-day operations.

## 5.2 Server Architecture

Since the application follows a complex structure in relation to the frontend, it must expose a handful amount of methods related to each object type, which will be showcased in the following sub-sections in a more structured manner, as well as their association with the server itself. The server architecture follows a version of the MVC(Model-View-Controller) design pattern. However, since the frontend is developed as a separate client application, the server-side implementation includes only the Model and Controller layers.

### 5.2.1 Database Design and Models

The proposed database management system used for this application's demo is the PostgreSQL, as it is standard compliant, feature-rich and extensible. Nevertheless the database used is irrelevant in future use, as every server that is being run to can define it's database in the configuration file in the .NET project, more specifically:

```
1 "ConnectionStrings": {  
2     "Default": "Server=localhost;Include Error Detail=true;  
3         Port=5432;User Id=postgres;  
4         Password=*****;Database=integration;"  
5 }
```

Listing 5.1: Defining the connection string to connect to the database

This thus being used in the Program.cs file, the entry point of the application:

```
1 builder.Services.AddDbContext<DatabaseContext>(options =>  
2     options.UseNpgsql(builder.Configuration  
3         .GetConnectionString("Default")!));
```

Listing 5.2: Creating a database context to create the connection to the database

A database context refers to an object that serves as a bridge between the application and database, especially in this case, there ORM(Object-Relational-Mapping) is used in Entity Framework, mentioned in the previous chapter's section 4.1, providing a methodology and mechanism for object-oriented systems to hold their long-term data safely in a database, with transactional control over it, yet have it expressed when needed in program objects [O'N08]. To present the Models, i provide the full database schema and after for each Object Type and will showcase, in this way, their characteristics:

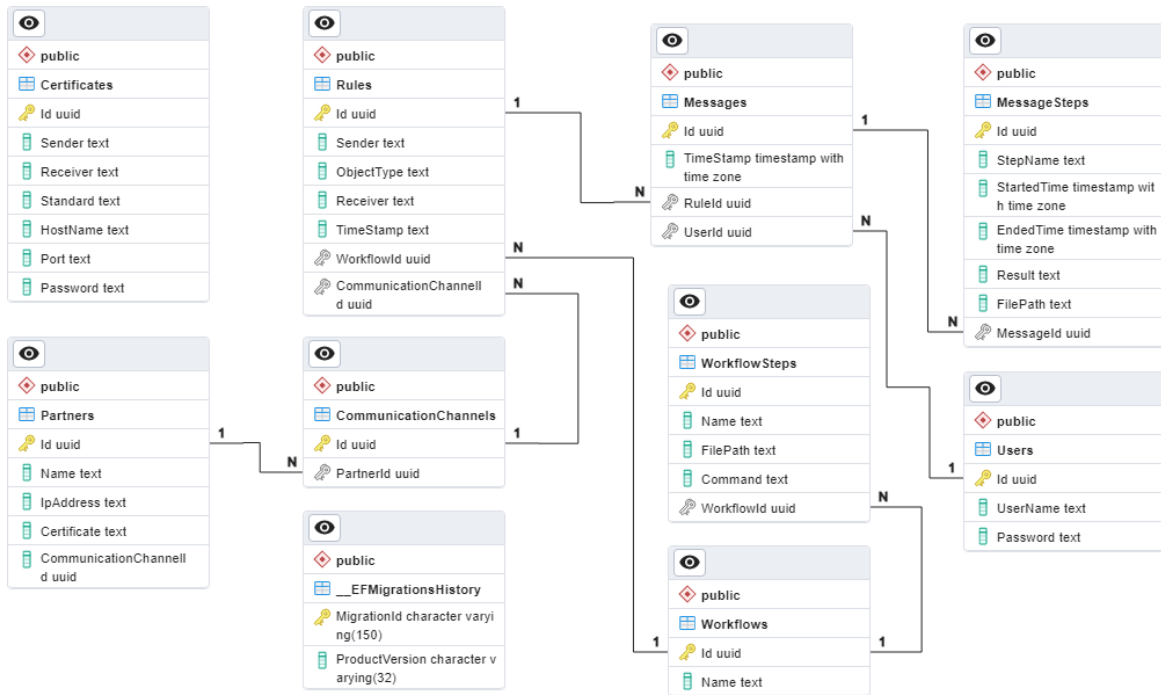


Figure 5.1: Entire database schema

To better understand the architecture and to have a grasp of the connections, all relevant objects will be introduced and explained in detail below:

- **User** As it stands, many companies that are keen to use this application should have not less than 2 employees, thus having a user assigned to them is essential. Here we are not working with e-mails, and user-defined parameters like birthday, name or occupation, we only need to differentiate users by their user-name, so that every employee can be assign their own tasks, and a password for security reasons. This password is generated by the application administrator, which in this case is only one: actis.

```

1 public class User
2 {
3     public Guid Id { get; set; }
4     public string UserName { get; set; } = string.Empty;
5     public string Password { get; set; } = string.Empty;
6     [JsonIgnore]
7     public ICollection<Message> Messages { get; set; } = null
8     ↪ !;
9 }

```

Listing 5.3: Code for creating the User model in .NET

As shown in the listing above, the user appears to have a username, a password and a collection of Messages. These messages can be assigned to an user via the following relation:

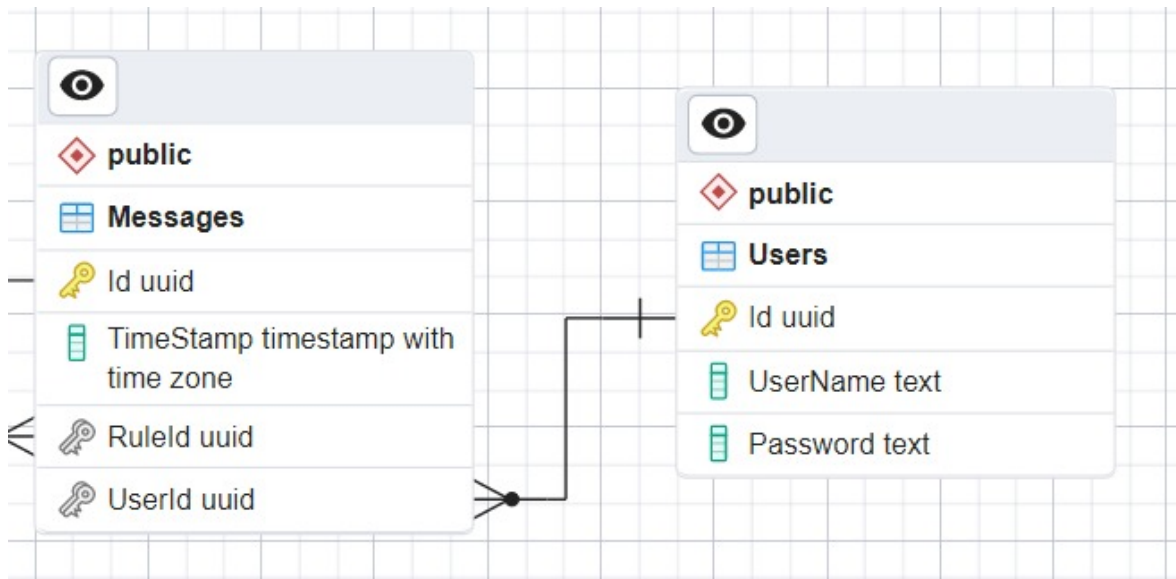


Figure 5.2: User-Message relation

Notice that 1:n relationship between User and Messages allows the listing 5.3 to happen. A user can be assigned multiple messages or none. Putting an emphasis on none, as the user itself can have a total of 0 assigned messages, all of them being assigned to *actis*(Admin user) at first.

- **Workflow & Workflow Step** Imagine a set of instructions a message has to follow in order to achieve it's goal, this is what a workflow is, in a nutshell. I considered having a workflow object to represent a collection of steps, hence creating the relation between Workflow and Workflow Step. A step itself is much more than an instruction, though, as far as it concerns the user, a workflow can go through the following defined-steps: COPY, SEND, CONVERT and REMOVE. Each of these will be covered in the future front-end section.

In the context of the Integration Manager application, this relationship allows messages or files to be processed according to configurable workflows. Administrators can define workflows dynamically and assign them to messages, with each step being recorded, monitored, or re-tried as needed.

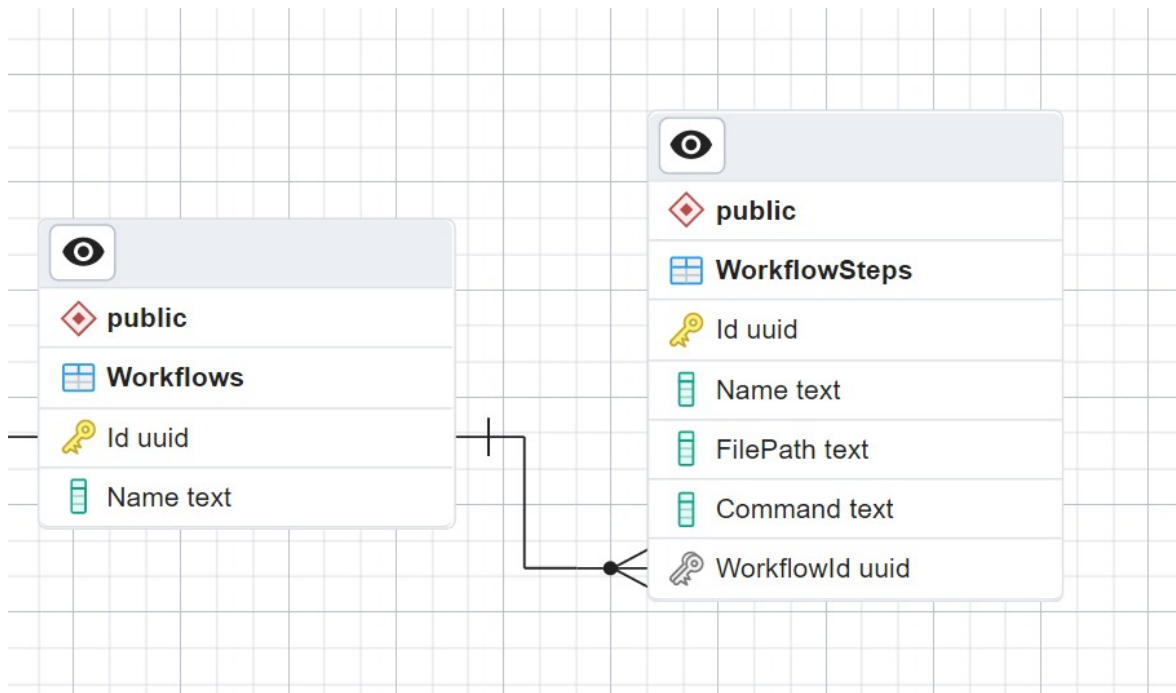


Figure 5.3: Workflow-Workflow Step relation

The workflow can be considered a key component in organizing this application, it revolves around maintaining an order of steps which needs to be processed, by providing these steps. A Workflow Step contains also a file path and a command field. The file path field is used for assigning powershell scripts for each user to upload and use during the message 'CONVERT' step. The command is not a user-defined command, but rather an automatic command which is generated in order to run the script itself.

- **Partner, Communication Channel & Certificate**

When working with customers, suppliers and different systems we need to establish a way of communication between them. This usually happens first via contract agreements by both parties, then by manual connection by mail, and ultimately, by integrating the partner in our system.

By defining a Partner entity, we can represent an external party integrated into the application. Each partner can communicate with the system through a defined Communication Channel. This ensures that every partner has a unique communication setup, which are all included in the Certificate object. Although it does not maintain an explicit relational link in this schema, it is used to store sensitive credentials or certificates associated with a partner, such as:

```

1 public class Certificate
2 {
3     public Guid Id { get; set; }
4     public string Sender { get; set; } = string.Empty;
5     public string Receiver { get; set; } = string.Empty;
6     public string Standard { get; set; } = string.Empty;
7     public string HostName { get; set; } = string.Empty;
8     public string Port { get; set; } = string.Empty;
9     public string Password { get; set; } = string.Empty;
10 }

```

Listing 5.4: Code for creating the Certificate model in .NET

So, when a partner wants to reach to us and have an agreement to create a connection with us, they shall provide us a certificate. The certificate contains a Sender, which, whether they are a supplier or customer, can be us or them, a Receiver, which should be the opposite party in the exchange. Whether the sender is a supplier or a customer, the receiver represents the intended recipient of the message or document, a standard, which, in our use case can only be ANSI X12 or EDIFACT. The certificate also contains a hostname, port and password which are used in fairness to connect to the partner's backend which is running on a different server than ours, the password ultimately being a JWT token used to verify the backend's endpoints and ensure security.

- **Rule** A rule is pretty self-explanatory, is the rule followed by the message in order to be processed. Rules play a critical role in managing the processing and routing of messages and documents. However, these rules cannot operate alone; their execution is fundamentally dependent on the availability and configuration of both communication channels and workflows. Here the Rule object is split based on fields and entities:

```

1 public class Rule
2 {
3     public Guid Id { get; set; }
4     public string Sender { get; set; } = string.Empty;
5     public string ObjectType { get; set; } = string.Empty;
6     public string Receiver { get; set; } = string.Empty;
7     public string TimeStamp { get; set; } = string.Empty;
8     ...

```

Listing 5.5: Code for creating the Rule model in .NET (Non-Key fields)

The Sender and Receiver refer to the Certificate's sender and receiver, and this is ultimately followed by the Object Type, referring back to 2.2. An object type does not require an entity itself as it can be a user-defined structure only to identify certain messages. For example Partner A (Sender) sends an invoice to Partner B (Receiver), the user which defines this rule can set the object type, `invoic`, `invoice`, `142` (ANSI X12 standard message type for invoices) or `PAPBin` (Partner A Partner B invoice), showcasing an open-minded way of choosing and asserting in the Integration Manager. The object type is irrelevant; for the sender, but very important for the receiver. The receiver knows what it is expecting but doesn't know what message type exactly.

```

1      ...
2      public Guid WorkflowId { get; set; }
3      public Workflow Workflow { get; set; } = null!;
4
5      public Guid CommunicationChannelId { get; set; }
6      [JsonIgnore]
7      public CommunicationChannel CommunicationChannel { get;
8          ↪ set; } = null!;
9
10     [JsonIgnore]
11     public ICollection<Message> Messages { get; set; } = null
12         ↪ !;
13 }

```

Listing 5.6: Code for creating the Rule model in .NET (Key fields)

Together, communication channels and workflows create an environment where rules can be executed contextually and effectively, enabling dynamic and automated management of complex business processes. Thus, any rule implementation relies on the precise integration of these two components to function correctly, ensuring that the system can respond appropriately to diverse scenarios and maintain operational integrity.

As mentioned, the rule acts as the central connection for messages to follow, the partner / communication channel means there is a connection to the other party via certificates. This connection ensures that all communication is secure, authenticated, and properly routed between trusted entities. Each partner is linked to a specific communication channel that defines the protocol and parameters used for message exchange, while certificates provide the necessary credentials to establish encrypted and verified sessions. Without this secure link, messages cannot be reliably transmitted or received, potentially

exposing the system to unauthorized access or data corruption. Moreover, the rule relies on predefined workflows that dictate the sequence of processing steps each message must undergo once it enters the system. These workflows coordinate tasks such as validation, transformation, storage, and notification, ensuring that messages progress through the system in an orderly and consistent manner. Together, the interplay between rules, communication channels, partners, certificates, and workflows forms the backbone of the application's message handling logic, enabling robust, secure, and efficient processing of all business-critical documents and data flows.

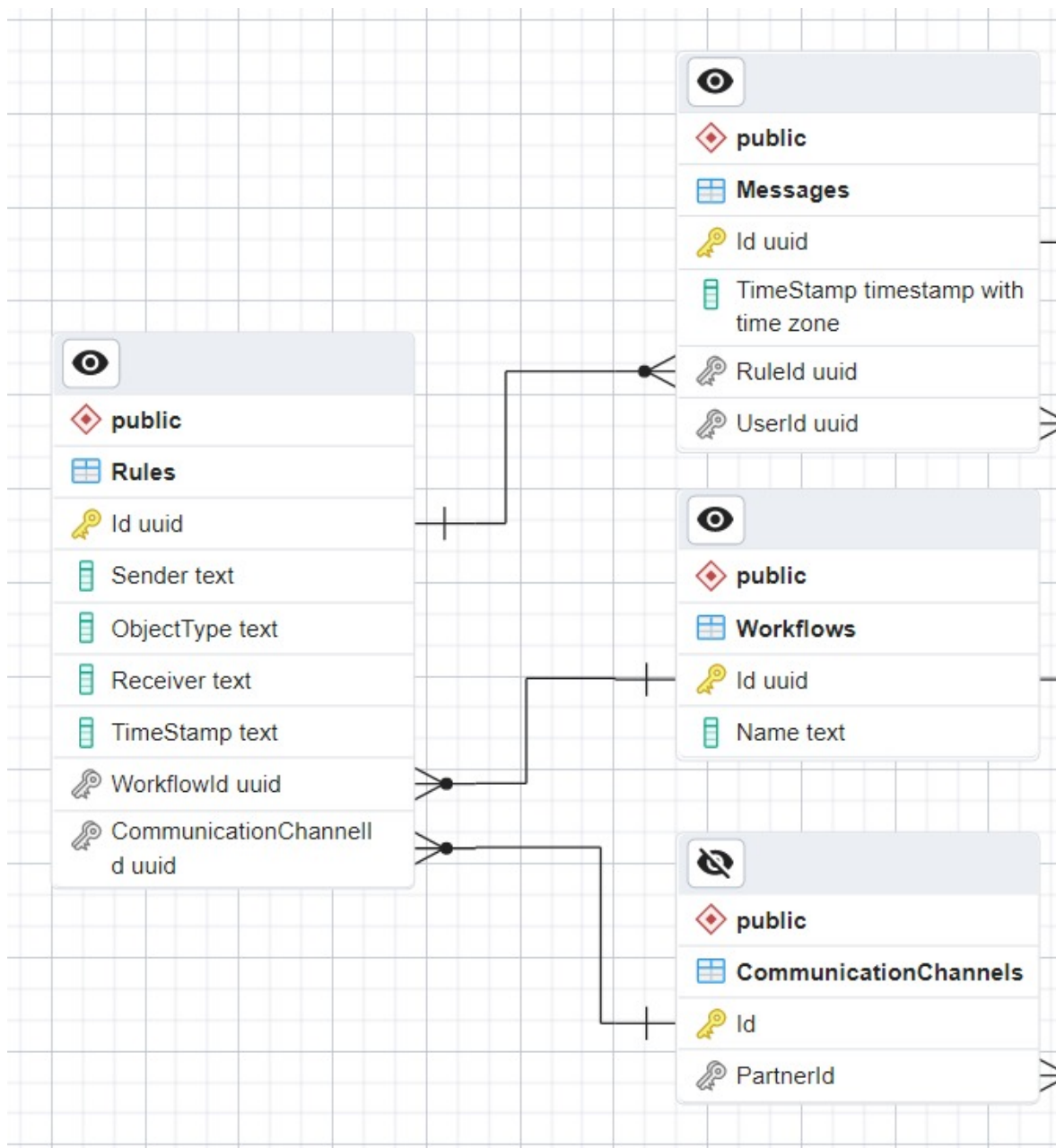


Figure 5.4: Rule relation to Message, Workflow & Communication Channel



- **Message & Message Step** Coming to the most important and relevant object in the whole application, the Message contains a pair of Message Steps, similar to the Workflow-Workflow Steps relation, the Message Step object reflects on the Workflow Step object, as the StepName field in Message Step is the Name field in Workflow Step:

```
1 public class MessageStep
2 {
3     public Guid Id { get; set; }
4     public string StepName { get; set; } = string.Empty;
5     public DateTime StartedTime { get; set; }
6     public DateTime EndedTime { get; set; }
7     public string Result { get; set; } = string.Empty;
8     public string FilePath { get; set; } = string.Empty;
9     public Guid MessageId { get; set; }
10    [JsonIgnore]
11    public Message Message { get; set; } = null!;
12 }
```

Listing 5.7: Code for creating the Message Step in .NET

Following a number of interesting fields, the Message Step consists entirely of organization fields, fields that provide information for the user and not for the application. For example:

- **Started Time** : the date and time of when the step has been started and is awaiting execution.
- **Ended Time** : the date and time of when the step has finalized executing.
- **Result** : result of the execution, can either be (READY - ready to execute, OK - executed successfully or ERROR - unsuccessful execution).
- **File Path** : after each execution the file may change, this field prevents of applying new changes to the same file i.e. I execute a step to convert the file to PDF, then I want to send the PDF converted file to the receiving party. This field allows to send the converted file, instead of the original file, think of it as to track the files.

Ultimately, each message being defined by a rule and multiple steps, the Message controls the base of the application, messages can be stored, converted, sent, removed and kept track of.

## 5.2.2 Repository Layer

As presented in sub-sector 5.2.1, the application consists of a fairly simple database with a total of 10 main objects, but it needs to be connected to the mainframe application. .NET provides a handful of options to manage a database, one of them being the Entity Framework Core, which is utilized as the Object-Relational Mapping (ORM) tool to facilitate interactions between the application and the underlying PostgreSQL database. I present an example of how the Workflow-Steps - Workflows - Rules relationship is made:

```

1 public class DatabaseContext : DbContext
2 {
3     ...
4     public virtual DbSet<Workflow> Workflows { get; set; } = null!;
5     public virtual DbSet<WorkflowStep> WorkflowSteps { get; set; }
6         ↪ = null!;
7     public virtual DbSet<Rule> Rules { get; set; } = null!;
8
9     protected override void OnModelCreating(ModelBuilder
10         ↪ modelBuilder)
11     {
12         ...
13         modelBuilder.Entity<Rule>()
14             .HasOne(r => r.CommunicationChannel)
15             .WithMany(c => c.Rules)
16             .HasForeignKey(r => r.CommunicationChannelId);
17
18         modelBuilder.Entity<Rule>()
19             .HasOne(r => r.Workflow)
20             .WithMany(w => w.Rules)
21             .HasForeignKey(r => r.WorkflowId);
22
23         modelBuilder.Entity<WorkflowStep>()
24             .HasOne(ws => ws.Workflow)
25             .WithMany(w => w.WorkflowSteps)
26             .HasForeignKey(ws => ws.WorkflowId);
27     }
28 }
```

Listing 5.8: Code for creating ORM connections to the database

At the start each entity is being created using **DbSet<Entity> EntityName**. This is a required structure in order to be able to signal Entity Framework Core to create the table **EntityName**. To facilitate the creation of a table which contains foreign key the following code should be implemented:

```
1 modelBuilder.Entity<ParentEntity>()
2     .HasOne(p => p.ChildEntity)
3     .WithMany(c => c.ParentEntity)
4     .HasForeignKey(p => p.ChildEntityId);
```

Listing 5.9: Code for creating a 1:n relationship

Note that **.HasOne(p => p.ChildEntity)** refers to the child, as well as **p => p.** refers to the parent. This code essentially encapsulates the idea for having the 1: relationship. By using **.WithMany(c => c.ParentEntity)**, the application handles this as the :n relationship, suggesting that the parent contains many child objects. Ultimately, the foreign key is assigned in **.HasForeignKey(p => p.ChildEntityId)**.

In order to create a table which does not contain foreign relationships, the following must be done:

```
1 modelBuilder.Entity<Certificate>()
2     .HasKey(c => c.Id);
```

Listing 5.10: Code for creating a non-related table

Here only the Id needs to be mentioned in order to set the primary key in the database.

In order to represent a 1:1 relation using Entity Framework Core's database context operation, this code should be implemented:

```
1 modelBuilder.Entity<Partner>()
2     .HasOne(p => p.CommunicationChannel)
3     .WithOne(c => c.Partner)
4     .HasForeignKey<CommunicationChannel>(p => p.PartnerId);
```

Listing 5.11: Code for creating a 1:1 relationship

It follows the same principle as the previous listing 5.9, but instead of **.WithMany(c => c.ParentEntity)** it uses **.WithOne(c => c.ParentEntity)** as showcased above using the Partner - Communication Channel relationship.

### 5.2.3 Controllers

In a .NET application, the controllers play a massive part in the management of the HTTP requests, their processing and providing adequate responses. They act as intermediates between the Model (database) and View (Client-Side), thus making sure that the business logic is applied correctly. [CSP12]. Hence, every controller follows the same constructor and start, this assimilation is creating a sense of togetherness, being said that every controller is alike to a point, generic:

```

1 [ApiController]
2 [Route("api/[controller]")]
3 public class TestController : ControllerBase
4 {
5     private readonly DbContext context;
6     private readonly IConfiguration configuration;
7
8     public TestController(DbContext context, IConfiguration
9         ↪ configuration)
10    {
11        this.context = context;
12        this.configuration = configuration;
13    }
14    ...
15 }

```

Listing 5.12: Code for creating the constructor for a controller

Controllers in this context are modules which define corresponding methods to each final API point. These methods manage different actions, for example, user registration, authentication, message sending and retrieval. Hence the application is structured in multiple controllers:

- **User / Authentication Controller** splits itself in 2 main parts. Users can be either Admin users or Normal users, with Admin users having a lot more control over the endpoints. This allows the application to ensure security as well as some sort of control over in the application, thus making it more entitled to avoid attacks. For example a normal user cannot create or delete users, as shown below:

```

1      [HttpPost("register")]
2      [Authorize(Roles = "Admin")]
3      public async Task<ActionResult<RegisterResponseDTO>>
4          ↪ RegisterAsync(AuthenticateUserDTO user)
5      {
6          ...
7      }

```

Listing 5.13: Register endpoint having Admin role authorization

Instead of having normal **[Authorize]** which allows every type of JWT token created by the application, **[Authorize(Roles = "Admin")]** indicates that a JWT token with Admin authorization must be provided in order to access this endpoint.

```

1      private string CreateJwtToken(User user)
2      {
3          var role = user.UserName.ToLower() == "actis" ? "Admin"
4              ↪ " : "User";
5
6          var claims = new[]
7          {
8              new Claim(JwtRegisteredClaimNames.Sub, user.
9                  ↪ UserName),
10             new Claim(JwtRegisteredClaimNames.Jti, Guid.
11                 ↪ NewGuid().ToString()),
12             new Claim("id", user.Id.ToString()),
13             new Claim("username", user.UserName),
14             new Claim(ClaimTypes.Role, role)
15         };
16
17         var key = new SymmetricSecurityKey(Encoding.UTF8.
18             ↪ GetBytes(this.configuration.GetSection("
19                 ↪ AppSettings:Key").Value!));
20         var creds = new SigningCredentials(key,
21             ↪ SecurityAlgorithms.HmacSha256);
22
23         ...
24     }

```

Listing 5.14: 1st part of method to generate JWT token for user

Notice that when creating claims, the application is required to use as an instance the user itself, in this example claims have been made the user's username, specifically when the user is 'actis' they are given the admin role.

```

1
2      ...
3      var token = new JwtSecurityToken(
4          claims: claims,
5          expires: DateTime.UtcNow.AddHours(2),
6          signingCredentials: creds
7      );
8
9      return new JwtSecurityTokenHandler().WriteToken(token)
10         ↪ ;
11     }

```

Listing 5.15: 2nd part of method to generate JWT token for user

Ultimately, the method returns the token as a string to be used in future endpoint calling, with an additional option to expire the token after 2 hours. This option is highly configurable depending on each system and the user experience.

One method only in the entire application does not contain the **[Authorize]** parameter, this being the **/login** endpoint, as shown below, using the built in SwaggerUI, the login endpoint does not contain a lock:

Authentication		^
POST	/api/Authentication/login	✓
GET	/api/Authentication/users	✓ 🔒
POST	/api/Authentication/register	✓ 🔒
DELETE	/api/Authentication/delete/{id}	✓ 🔒

Figure 5.5: Authentication Controller's endpoints

As shown in the picture above GET method for requesting the users **/users** is used for assigning users, authorized normally (no admin needed), whilst POST method **/register** and DELETE method **/delete/{id}** require admin authorization for calling.

- **CRUD controllers** are taking the most part in any controller, basic operations needed for managing a database are essential to any application. Hence I will provide a detailed specification of the **Partner** controller as a template for most controllers in the application:

Partner	
GET	/api/Partner/partners/{name}/{ipAddress}/{certificate}
GET	/api/Partner/partners/{id}
POST	/api/Partner/partner/add
PUT	/api/Partner/partner/update/{id}
DELETE	/api/Partner/partner/delete/{id}

Figure 5.6: Partner Controller endpoints

The GET method `/partners/{id}` is expected to return one partner as the id's are unique, hence in the frontend when selecting a partner this method is called automatically to provide information about one partner:

```

1      [HttpGet("partners/{id}")]
2      [Authorize]
3      public async Task<ActionResult<List<Partner>>>
4          ↪ GetPartnerById(string id)
5      {
6          var partner = await this.context.Partners.
7              ↪ FirstOrDefaultAsync(p => p.Id.ToString() == id);
8          if (partner == null)
9          {
10             throw new Exception("There are no partners");
11         }
12         return Ok(partner);
13     }

```

Listing 5.16: Method for getting a partner by id

Although separate validation also takes place, a little bit more security never hurts, by only checking if the partner exists in lines 6-9 we provide the user a message in case of any doubts.

The method `/partners/{name}/{ipAddress}/{certificate}` is expected to return all of the partners that match these parameters. This application is based on efficiency and when handling with multiple partners sometimes it can be easier to filter them out using relevant fields:

```

1      [HttpGet ("partners/{name}/{ipAddress}/{certificate}")]
2      [Authorize]
3      public async Task<ActionResult<List<Partner>>> GetPartners
4          ↪ (string name, string ipAddress, string certificate)
5      {
6          var namePattern = name.Replace('*', '%');
7          var ipAddressPattern = ipAddress.Replace('*', '%');
8          var certificatePattern = certificate.Replace('*', '%')
9          ↪ ;
10
11         var partners = await this.context.Partners.Where
12             (p => EF.Functions.Like(p.Name, namePattern) &&
13                 EF.Functions.Like(p.IpAddress, ipAddressPattern)
14                 ↪ &&
15                 EF.Functions.Like(p.Certificate,
16                 ↪ certificatePattern))
17             .ToListAsync();
18
19         if (partners == null)
20         {
21             throw new Exception("There are no partners");
22         }
23
24         return Ok(partners);
25     }

```

Listing 5.17: Method for getting partners using different filters

In this method, the patterns are configured in the way that a Linux machine operates, for example when providing the **name** parameter as **nissan\*** this would return all partners with a name matching this pattern (nissan, nissanMX, nissantst, etc.).



The POST method **/partner/add**, takes as parameter a PartnerDTO, a lot of DTO's are present in the application as they are the starting building block of POST and PUT methods:

```

1      [HttpPost("partner/add")]
2      [Authorize]
3      public async Task<ActionResult<Partner>> AddPartner([
4          ↪ FromForm] PartnerDTO partner)
5      {
6          var actualPartner = await this.context.Partners.
7              ↪ FirstOrDefaultAsync(p => p.IpAddress == partner.
8              ↪ IpAddress);
9
10         if (actualPartner != null)
11         {
12             throw new Exception("There already exists a
13                 ↪ partner with this IP Address");
14         }
15
16         var addPartner = new Partner
17         {
18             Id = new Guid(),
19             Name = partner.Name,
20             IpAddress = partner.IpAddress,
21             Certificate = partner.Certificate
22         };
23
24         this.AddCertificate(partner.CertificateFile);
25         this.CopyCertificate(partner.CertificateFile);
26
27         await this.context.Partners.AddAsync(addPartner);
28         await this.context.SaveChangesAsync();
29
30         return Ok(addPartner);
31     }

```

Listing 5.18: Method for adding a partner object using a partner DTO

A DTO, in this context, has the same overall structure as the normal entity, except the ID, which is generated in the method itself, the user cannot provide the ID since is in the GUID format and also because it is not user-friendly and overall not necessary. Throughout the method, we can notice the same

validation patterns being used as we do not want to create a partner with the same IP address as another already existing partner.

A partner is created and its certificate is provided in the DTO, hence the method `this.AddCertificate(partner.CertificateFile)`, and also the method `this.CopyCertificate(partner.CertificateFile)` are called. These methods add the certificate to the database and saves the certificate file in a local folder.

The PUT method `/partner/update/{id}` works seamlessly the same as the POST method mentioned above, but instead of adding a partner, the method searches for a partner via the id provided in the query, if it finds it it will update that partner's fields accordingly:

```

1      ...
2      var actualPartner = await this.context.Partners.
      ↪ FirstOrDefaultAsync(p => p.IpAddress == partner.
      ↪ IpAddress);
3      ...
4      actualPartner.Name = partner.Name;
5      actualPartner.IpAddress = partner.IpAddress;
6      actualPartner.Certificate = partner.Certificate;
7      ...

```

Listing 5.19: Method for updating a partner object using a partner DTO

The DELETE method `/partner/delete/{id}` follows the same principle as the PUT method, first it searches for available partners which exists and have the id, since the id is unique it will only need to return one entity.

```

1      ...
2      var actualPartner = await this.context.Partners.
      ↪ FirstOrDefaultAsync(p => p.IpAddress == partner.
      ↪ IpAddress);
3      ...
4      this.context.Certificates.Remove(certificate);
5      ...

```

Listing 5.20: Method for deleting a partner object using the id

- **Message Controller** follows a different approach as it contains way more methods than the CRUD operations, hence the Message controller is exposing 10 methods:

Message	
GET	/api/Message/messages/{pattern}
GET	/api/Message/message/{id}
POST	/api/Message/message/step/file
PUT	/api/Message/message/user/assign/{id}
GET	/api/Message/messages/out/check
GET	/api/Message/messages/in/check
POST	/api/Message/message/store
POST	/api/Message/message/restart
PUT	/api/Message/message/process/step/{id}
DELETE	/api/Message/message/delete/{id}

Figure 5.7: Message Controller endpoints

Besides the usual CRUD operations, the Message controller is showcasing 6 more methods, 2 of them referring to processing the message by steps, and 4 of them regarding the file storage and management system.

To start off with, the Message controller acts also as the Message Step controller, hence it exposes the most complex method in the whole application. I will present it in parts in order to create a general overview of it's functionalities:

The first part of the method showcases the extraction of the message step, as the Id is being retrieved but also all of the underlying entities (Communication Channel, Partner, Message, Rule and Workflow), all need to be included in the search, as they are an important part of this current method:

```

1  var messageStep = await this.context.MessageSteps.Include(
    ↪ messageStep => messageStep.Message)
2      .ThenInclude(message => message.Rule).ThenInclude(
    ↪ rule => rule.CommunicationChannel)
3      .ThenInclude(communicationChannel =>
    ↪ communicationChannel.Partner)
4      .Include(messageStep => messageStep.Message) .
    ↪ ThenInclude(message => message.Rule)
5      .ThenInclude(rule => rule.Workflow).ThenInclude(
    ↪ workflow => workflow.WorkflowSteps) .
    ↪ FirstOrDefaultAsync(ms => ms.Id.ToString()
    ↪ == id);

```

Listing 5.21: Extracting a message step

As there are 4 steps defined in the application, each of them need to have a set of actions defining them:

```

1  try
2      {
3          var relativeFilePath = CopyFile(filePath, fileName
    ↪ );
4
5          messageStep.Result = "OK";
6          messageStep.EndedTime = DateTime.UtcNow;
7          messageStep.FilePath = relativeFilePath;
8          await this.context.SaveChangesAsync();
9
10         await this.PrepareNextStep(messageStep,
    ↪ relativeFilePath);
11         await this.context.SaveChangesAsync();
12     }
13     catch (Exception ex)
14     {
15         messageStep.Result = "ERROR";
16         messageStep.EndedTime = DateTime.UtcNow;
17         await this.context.SaveChangesAsync();
18     }
19     break;

```

Listing 5.22: Copy step

The copy step follows a simple procedure: save the file and prepare the next step, a common pattern occurs here for all steps, because at each step, the file needs to be saved and the next step must be prepared.

```
1      try
2      {
3          await this.DeleteMessage (messageStep.MessageId.
4              ↳ ToString());
5      }
6      catch (Exception ex)
7      {
8          messageStep.Result = "ERROR";
9          messageStep.EndedTime = DateTime.UtcNow;
10         await this.context.SaveChangesAsync();
11     }
12     break;
```

Listing 5.23: Remove step

The remove step, unlike all other steps does not have the need to prepare the next step as the remove step is always the last, hence this removes the message, but not the file.

```
1      var psi = new ProcessStartInfo
2      {
3          FileName = "powershell.exe",
4          Arguments = $"-ExecutionPolicy Bypass -File \"{
5              ↳ scriptPath}\" -FilePath \"{filePath}\"",
6          RedirectStandardOutput = true,
7          RedirectStandardError = true,
8          UseShellExecute = false,
9          CreateNoWindow = true
10     };
11     ;
```

Listing 5.24: Shell step

The convert step reflects on the type of machine the server is running, at the moment the application allows only powershell scripting, but this will be an issue to be faced in the future works of the application.

The send step ultimately allows redirection in the application, let's say I want to apply multiple scripts, I can create multiple rules and redirect the file to other rules based on the contents of the message.

### 5.2.4 Services

The service layer acts as a middleware between the repository and the controller in many web-based applications. Well in this case, it acts as the main part of each message processing controller, thus showcasing 2 Services:

**The DefaultMessageService** is an automated background service designed to continuously monitor and process message steps that follow a predefined, automatic processing rule — specifically, those where the rule timestamp is set to "DEFAULTS". Its primary purpose is to handle multi-step message workflows without manual user intervention, ensuring smooth and uninterrupted processing of integration messages.

This service operates within the hosted environment of an ASP.NET Core application by extending the BackgroundService class. Upon execution, it initiates a scoped DatabaseContext to retrieve pending message steps from the database. These steps are filtered and sorted based on their associated rules and their start time to ensure sequential execution. For each message step, the service dynamically constructs a filename based on rule metadata (such as sender, receiver, and object type), determines whether the current step is of type CONVERT or SHELL (which affects file suffixing), and prepares the payload for transmission.

The message step is then submitted via an authenticated HTTP PUT request to an internal endpoint. This endpoint is responsible for processing the step and returning an updated MessageStep object. If the response is successful, the next step in the sequence inherits the potentially modified file path, preserving continuity across the workflow. In the event of an error — either due to an HTTP failure or an exception — the rule's timestamp is updated to "PRE-DEFAULT" to flag the step as failed and halt further automatic processing. This mechanism ensures traceability, error detection, and fail-safe handling within the automation layer.

This service plays a critical role in the broader architecture by enabling headless, rule-based message orchestration. It integrates database querying, step management, file handling, and API communication in a clean loop that executes every five seconds. By doing so, it acts as the core automation engine for processing messages with default behavior — freeing users from manual input and ensuring that consistent operational logic is followed across all system components.

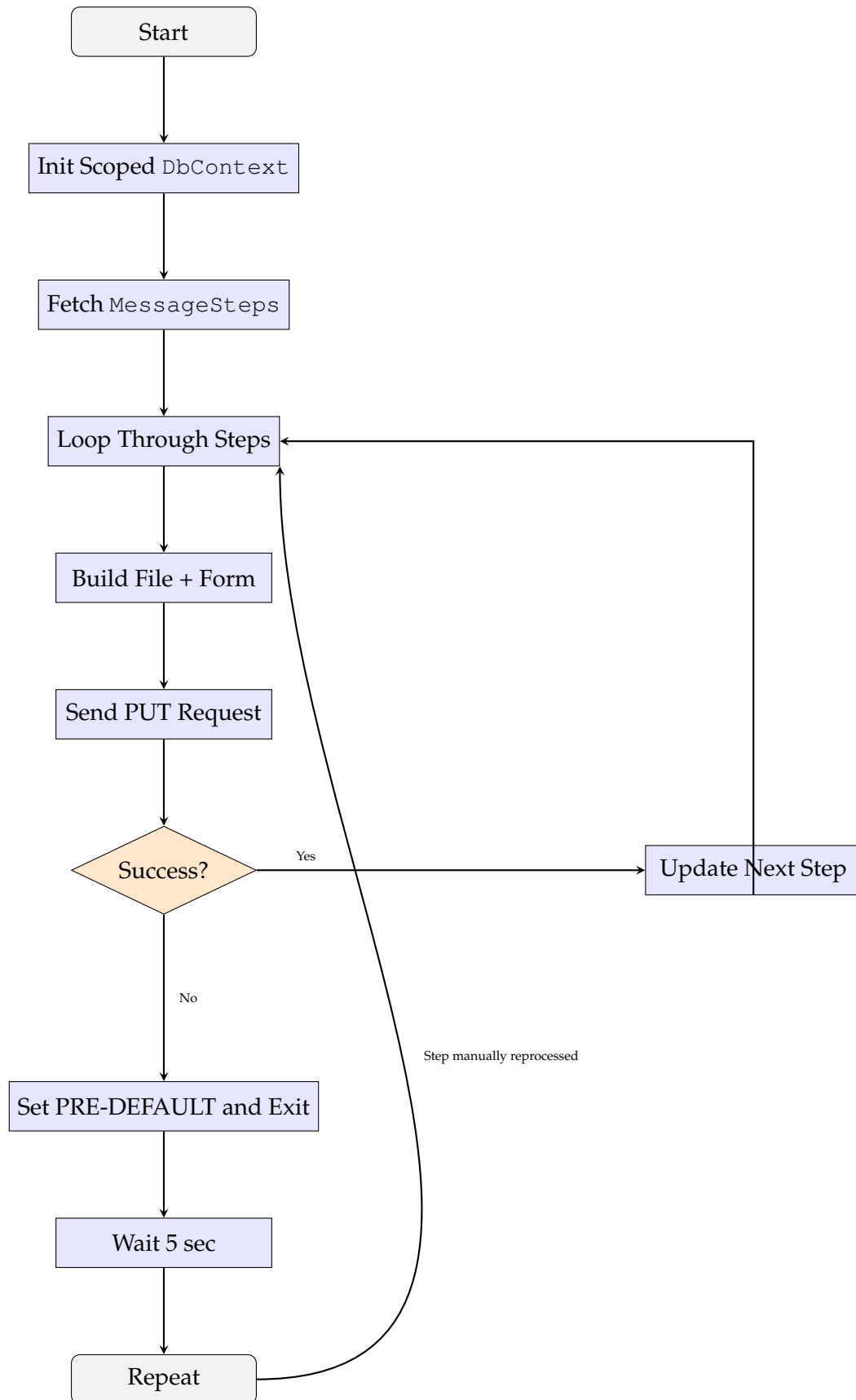


Figure 5.8: Refined flowchart of the DefaultMessageService

**The FolderWatchService** is a background task within the integration system that plays a critical role in automating inbound and outbound file handling. It continuously monitors two local directories: `archive/in` and `archive/out`. Each folder is dedicated to a specific type of communication — the former for incoming messages from external partners, and the latter for outbound messages that the system should dispatch. These folders serve as communication endpoints in file-based EDI or integration workflows.

When a file is detected in either folder, the service processes it asynchronously using the method `ProcessFileAsync`. This function is responsible for reading the file, determining the format (either EDIFACT or ANSI X12), and extracting key metadata such as sender, receiver, and object type. Depending on the format, different parsing logic is applied to correctly identify the values embedded in the message headers (ISA, ST, UNB, UNH, etc.). This metadata is then used to locate the appropriate Rule and Certificate from the database. These two entities represent the logical routing configuration and the authentication layer respectively, both of which are essential for secure message transmission.

Once the appropriate rule and certificate are located, the file is uploaded to the target endpoint via an HTTP POST request. The `SendOutMessage` and `SendInMessage` methods encapsulate this logic. These methods build a `MultipartFormDataContent` payload and send the request using an `HttpClient` configured with a bearer token derived from the certificate. After a successful upload, the file is deleted from the folder to avoid duplication and to maintain a clean state. In case of errors — such as inability to delete a file or upload failures — appropriate logging is performed, ensuring traceability and allowing for reprocessing if needed.

From a design perspective, the `FolderWatchService` encapsulates a polling-based integration mechanism. It abstracts the complexity of partner communication behind file-drop conventions and transforms those files into authenticated, API-based transmissions. This approach enables legacy compatibility (through file watching) while still benefiting from modern HTTP-based delivery mechanisms. Furthermore, this service is modular and reusable, making it straightforward to extend or adapt to new formats, rules, or destinations.



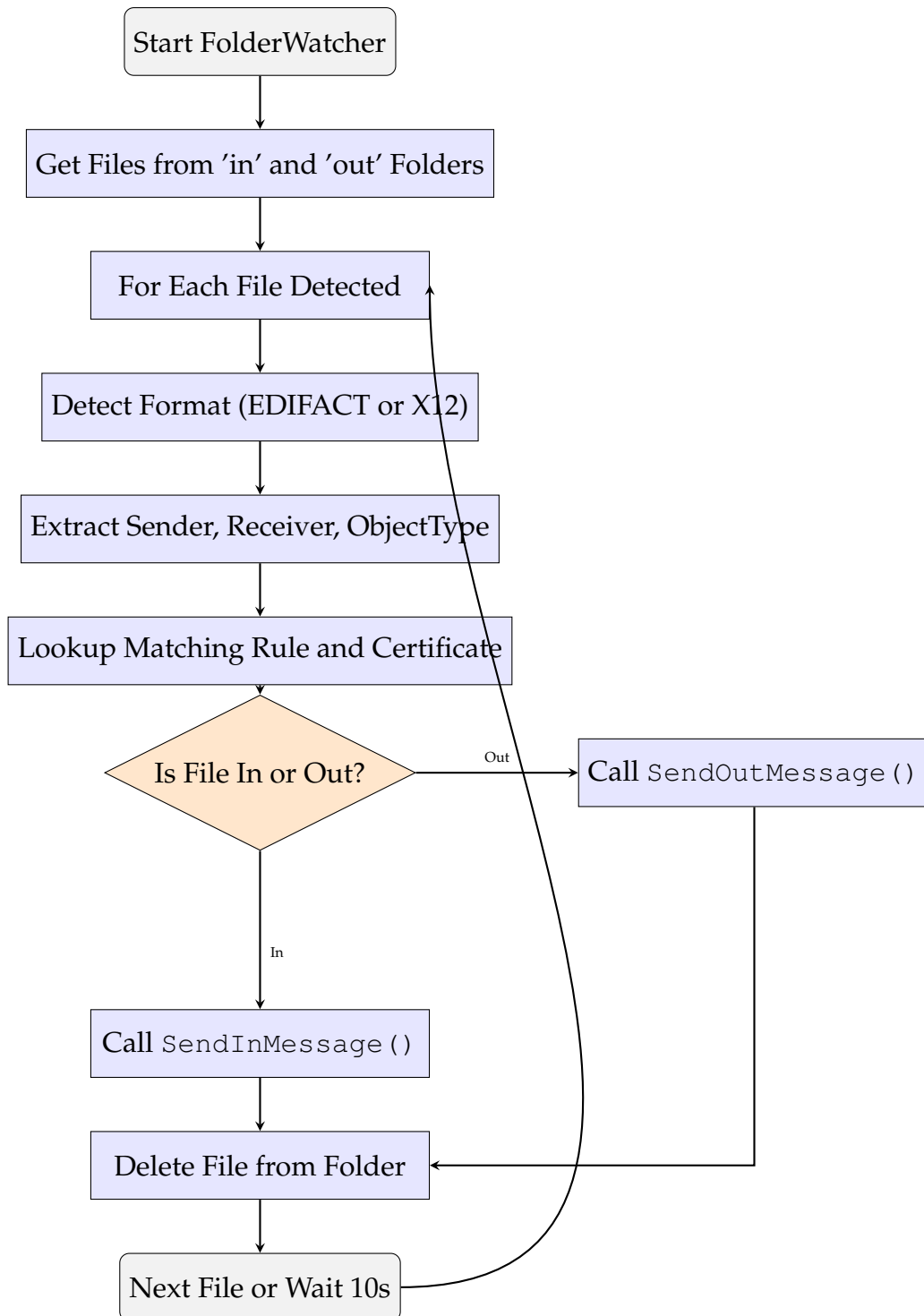


Figure 5.9: Execution flow of the FolderWatchService with inbound and outbound file handling

### 5.3 Web Interface Design

A client in client/server system is a crucial component which requests services from the server. This configuration forms a complete system with distinct roles for server

and client. Clients act as consumers, requesting services or information, whilst the server acts as the producers, to fulfill these requests. This model uses the capabilities of composite networking from numerous powerful workstations and a few dedicated servers. It quickly became popular, due to its hardware's accessibility and recognizing the fact that monolithic applications fail when the number of users or characteristics is heavily increasing. [Lew98]

### 5.3.1 Authentication

Whilst working with sensible data, we need to create a way for users to authenticate themselves to their corresponding server system. Unlike an usual client/server application, this application is designed to be used by all parties, thus creating a network of utilization. A user doesn't simply login, with a username and password, it also needs to connect to a specific server, where the employee operates its work. The user needs to provide also the hostname and the port, alongside the username and password, in order to complete a login successfully, as shown below:

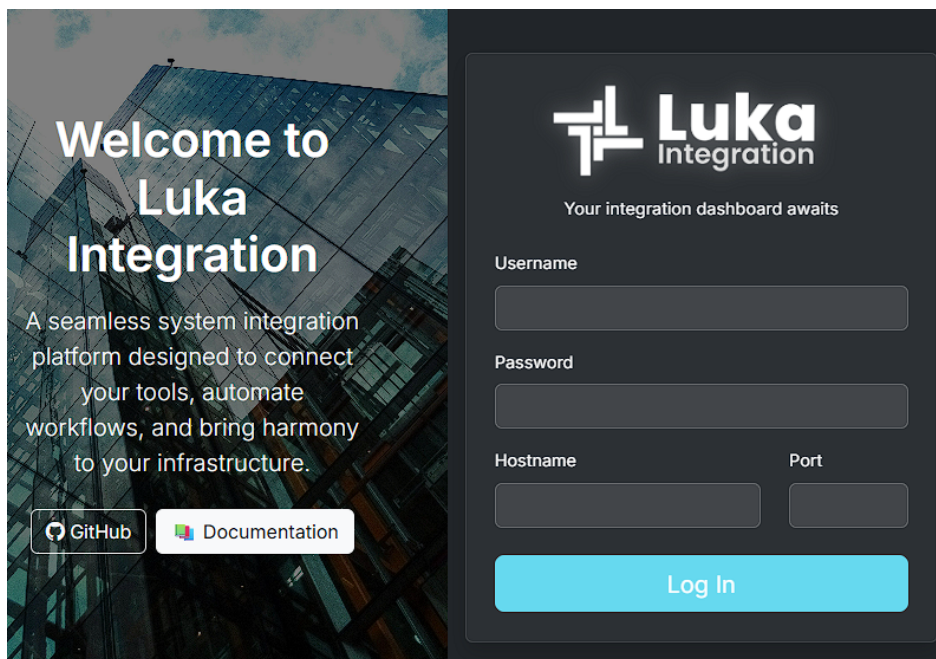


Figure 5.10: Landing page, for logging in

Notice that the landing page does not contain a register option, which might be a hassle at first, but in this case, only the administrator can provide the employee with login credentials. As mentioned previously in 5.2.1:

### 5.3.2 API Communication

Angular handles API communication primarily through its built-in `HttpClient` module, which is part of the `@angular/common/http` package. This module provides a streamlined and powerful way to send HTTP requests and receive responses from backend servers, typically RESTful APIs. Using `HttpClient`, Angular applications can perform various HTTP methods such as GET, POST, PUT, DELETE, etc., all while leveraging RxJS Observables to handle asynchronous data streams.

Once the login button is pressed the `login()` method in the Login component is called. This provides a validation form for missing fields, and calls the service layer underneath in order to send a requests to the server, it also saves the URL to which the user needs to connect using the hostname and port:

```
23  login(){
24      const error = this.validateForm();
25      if (error != '') {
26          alert(error); // Stop execution if form validation fails
27      }
28
29      this.service.login(this.user.userName, this.user.password, this.user.hostName, this.user.port).subscribe(data => {
30          localStorage.setItem('user', data.userName.toString());
31          localStorage.setItem('token', data.token.toString());
32          localStorage.setItem('userId', data.id.toString());
33          localStorage.setItem('myURL', `https://${this.user.hostName}:${this.user.port}/api`);
34          this.router.navigateByUrl("/");
35      }, (error: any) => {
36          console.log("Error object:", error);
37          console.log("Error.error:", error.error);
38
39          if (typeof error.error === 'string') {
40              if (error.error.includes("User doesn't exist")) {
41                  alert("User doesn't exist");
42              } else if (error.error.includes("Password is incorrect")) {
43                  alert("Password is incorrect");
44              } else {
45                  alert(`Server https://${this.user.hostName}:${this.user.port} is offline`);
46              }
47          } else {
48              alert(`Server https://${this.user.hostName}:${this.user.port} is offline`);
49          }
50      }
51      );
52  }
```

Figure 5.11: Code to call the service

```

14  login(userName: string, password: string, hostName: string, port: string) : Observable<any> {
15      const user = {
16          userName: userName,
17          password: password
18      };
19
20      const headers = new HttpHeaders({ 'Content-Type': 'application/json' });
21      return this.http.post<any>(`https://${hostName}:${port}/api/Authentication/login`, user, { headers });
22  }

```

Figure 5.12: Code to call the API

In this example the UserDTO is created using only the username and password, the only header needed is the **Content-Type** header as, no authorization is needed for the login API.

If the request was successful, the application the username, the user ID, the URL to where the API's routes are exposed, and the JWT token which should be used in the future:

```

226      const token = localStorage.getItem('token');
227      const headers = new HttpHeaders()
228          .append('Authorization', `Bearer ${token}`)
229          .append('Content-Type', 'application/json');

```

Figure 5.13: Example of headers which API route needs authorization

### 5.3.3 Components Overview

The components are structured in layers, as previously mentioned in 5.3.2, therefore it is structured into four main functional hierarchies: **Configuration**, **Login**, **Service**, and **Home**. This modular design promotes separation of concerns and ensures maintainability, scalability, and clear data flow within the application.

The **Configuration** section contains all administrative and model-related functionality necessary for defining core business structures. This includes components such as models, messages, rules, partners, workflows, and users. These elements collectively allow system administrators to establish and modify the business logic and operational framework of the application.

The **Service** hierarchy is responsible for handling all backend communication. It encapsulates API calls and related data models, abstracting HTTP interactions into

clean service layers. This ensures a consistent and type-safe interface for retrieving and managing data across the application.

The **Login** section manages user authentication and access control. It includes authentication mechanisms and route guards, ensuring that only authorized users can access specific parts of the application. This provides a secure entry point into the system and enforces user permissions effectively.

Finally, the **Home** module represents the user workspace, where operational tasks are executed. It contains features related to message processing and monitoring, giving users visibility into application activity and allowing for direct interaction with runtime data.

Together, these modules create a coherent system in which responsibilities are clearly defined, enabling both users and developers to interact with the application efficiently and securely.

### 5.3.4 UI Logic

The application's user interface (UI) logic is intricately woven together using Angular's powerful routing mechanism combined with route guards to enforce both authentication and authorization policies.

Routes serve as the backbone for navigation and view rendering in the app. Each route maps a URL path to a specific component responsible for displaying the appropriate content or functionality. This modular routing design helps keep the UI organized and maintainable.

To ensure security and a seamless user experience, route guards are applied to these routes. Guards act as gatekeepers that determine whether a user is permitted to navigate to a given route based on the application's authentication and authorization state.

For instance, the `authenticationGuard` is applied to most routes, including the home page, workflows, partners, rules, messages, and certificates. This guard verifies if the user is logged in by calling the authentication service. If the user is not authenticated, the guard redirects them to the login page, preventing unauthorized access to protected resources.

```
5  ✓ export const authenticationGuard: CanActivateFn = () => {  
6      if(inject(AuthService).isLoggedIn()){  
7          return true;  
8      }  
9      return inject(Router).navigateByUrl("/login");  
10  };
```

Figure 5.14: Guard for authorizing all users

In addition to authentication, role-based authorization is enforced through the `adminGuard`. This guard restricts access to user management routes (`/users` and `/user/register`) exclusively to the administrator user (username `=== 'actis'`). By checking the username stored locally along with the logged-in state, it ensures that only privileged users can manage other users, reinforcing security and maintaining proper access control.

```
12  ✓ export const adminGuard: CanActivateFn = () => {  
13      const authService = inject(AuthService);  
14  
15      const isLoggedIn = authService.isLoggedIn();  
16      const username = localStorage.getItem('user');  
17  
18      if (isLoggedIn && username === 'actis') {  
19          return true;  
20      }  
21  
22      return inject(Router).navigateByUrl("/login");  
23  };
```

Figure 5.15: Guard for authorizing admin user

In this application, Angular's routing system serves as the central mechanism that orchestrates how users interact with different parts of the UI. Each user interaction that leads to a new view—whether it's creating a workflow, configuring a

partner, or registering a user—is handled through the Angular Router, which maps URLs to their corresponding components.

This design provides a clear separation of concerns. Each feature (e.g., workflows, partners, rules) is encapsulated within its own component and accessible via a specific route. This modular approach not only improves maintainability but also makes the UI scalable as new features can be added without affecting existing logic.

```

22  export const routes: Routes = [
23    {path: 'login', component: LoginComponent},
24    {path: '', component: HomeComponent, canActivate: [authenticationGuard]},
25
26    {path: 'workflows', component: WorkflowComponent, canActivate: [authenticationGuard]},
27    {path: 'workflow/create', component: WorkflowDeclarationComponent, canActivate: [authenticationGuard]},
28    {path: 'workflow/configure/:id', component: WorkflowDeclarationComponent, canActivate: [authenticationGuard]},
29
30    {path: 'partners', component: PartnerComponent, canActivate: [authenticationGuard]},
31    {path: 'partner/create', component: PartnerDeclarationComponent, canActivate: [authenticationGuard]},
32    {path: 'partner/configure/:id', component: PartnerDeclarationComponent, canActivate: [authenticationGuard]},
33
34    {path: 'rules', component: RuleComponent, canActivate: [authenticationGuard]},
35    {path: 'rule/create', component: RuleDeclarationComponent, canActivate: [authenticationGuard]},
36    {path: 'rule/configure/:id', component: RuleDeclarationComponent, canActivate: [authenticationGuard]},
37
38    {path: 'message/create', component: MessageDeclarationComponent, canActivate: [authenticationGuard]},
39    {path: 'message/:id', component: MessageProcessingComponent, canActivate: [authenticationGuard]},
40
41    {path: 'certificate/generate', component: CertificateDeclarationComponent, canActivate: [authenticationGuard]},
42
43    {path: 'users', component: UserComponent, canActivate: [adminGuard]},
44    {path: 'user/register', component: UserDeclarationComponent, canActivate: [adminGuard]}
45  ];

```

Figure 5.16: All routes in the frontend application

## 5.4 Integration Flow

This is a typical end-to-end flow that users can follow within your application to set up and run a complete integration scenario after logging in:

### 5.4.1 Partner

In fairness, all partners need to be communicated before a connection shall be established, they should provide a certificate, but in case they do not, the application allows you to create your own certificate, to skip all the processes as follows:

## Certificate Generation

**Sender**

**Receiver**

**Standard**

**Hostname**

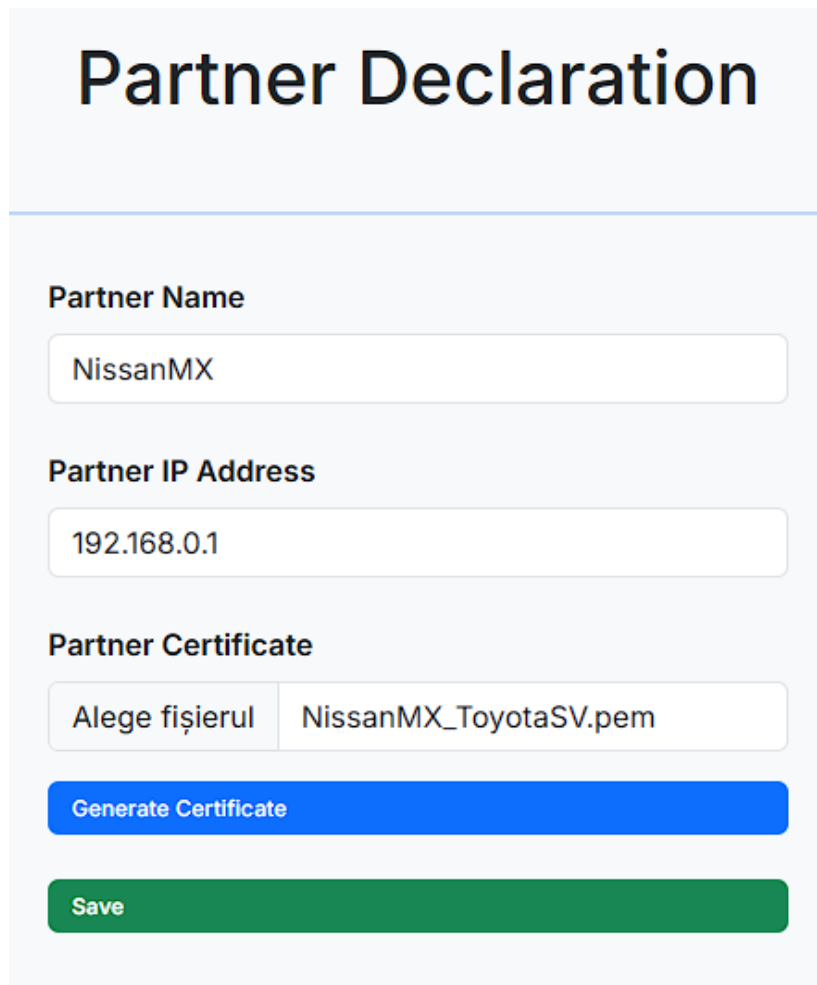
**Port**

Save

Figure 5.17: Creating a certificate

A certificate holds the basic information needed to connect and communicate securely between two systems. It includes who is sending the data, who is supposed to receive it, the technical standard being used, and where the data should go — defined by a hostname and port. Think of it as a digital envelope that helps ensure the message is sent in the right format to the right place.



The image shows a web form titled "Partner Declaration". It has three main sections: "Partner Name" with a text input field containing "NissanMX"; "Partner IP Address" with a text input field containing "192.168.0.1"; and "Partner Certificate" with a label "Alege fișierul" and a text input field containing "NissanMX\_ToyotaSV.pem". Below these fields are two buttons: a blue "Generate Certificate" button and a green "Save" button.

## Partner Declaration

**Partner Name**

**Partner IP Address**

**Partner Certificate**

Alege fișierul

**Generate Certificate**

**Save**

Figure 5.18: Creating a partner

A partner represents another company or system you exchange data with. Each partner has a name, an IP address that identifies their location on the network, and a certificate that defines how to securely communicate with them. This setup makes it easy to manage multiple connections while keeping everything secure and organized.

### 5.4.2 Workflow

The Workflow interface defines a complete process made up of multiple steps, where each step performs a specific action or handles a particular part of the operation. A workflow has a unique id and a descriptive name, helping identify it in larger systems.

## Workflows

Workflow Name

### Available Steps

COPY

SHELL

SEND

REMOVE

### Your Steps

COPY ×

SHELL ×

Selected: powershell.ps1

SEND ×

REMOVE ×

Figure 5.19: Creating a workflow

Each step is pre-defined and can be dragged and dropped to the user's liking, COPY and REMOVE are storage related, whilst SHELL and SEND are rule related. SHELL step provides an option to include a powershell script, for which the script will run on the message, all while the SEND step will send the message to the correct party by searching the certificates and finding in the message itself the corresponding SENDER ID and RECEIVER ID. Either way, the message will reach the party, but may not be in the intended rule. For example:

**ISA\*00\* \*00\* \*ZZ\*NissanMX\*ZZ\*ToyotaSV\*240520\*1435\*U\*00401\*0000\*0\*T\*:**

This is a ANSI X12 standard EDI message, previously we have generated certificate using these ID's (NissanMX & ToyotaSV), hence the system will know that this message will belong to which system.

### 5.4.3 Rule

In the context of EDI, a rule ensure that the data exchanged adheres to agreed-upon standards and business requirements, such as format compliance, mandatory fields, or specific transaction types.

## Current Selections:

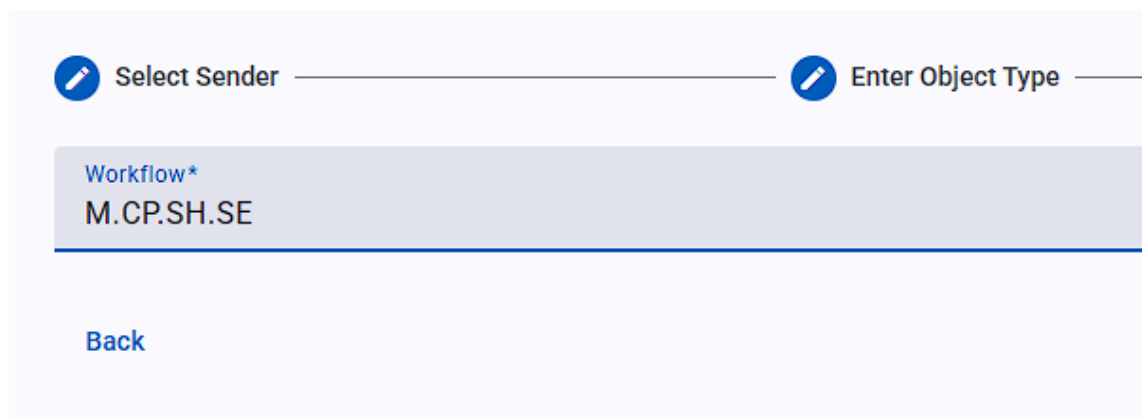
Sender: NissanMX

Object Type: 830

Receiver: ToyotaSV

Time Window: BANKTEST

Workflow: M.CP.SH.SE



The screenshot shows a web-based interface for creating a rule. At the top, there are two tabs: 'Select Sender' and 'Enter Object Type', both with a pencil icon. Below the tabs, there is a light blue box containing the text 'Workflow\*' and 'M.CP.SH.SE'. Below this box, there is a 'Back' button.

Figure 5.20: Creating a rule

Each message has a sender, a receiver and a object type, as well as a workflow and time window. An object type refers to the message type, in usual cases it is located inside the message, thus making it easier for the application to recognize the rule in which the message needs to be processed.

A time window is represented in 2 ways, either the message can be processed step-by-step or all steps automatically. I have pre-defined 2 time windows: BANKTEST and DEFAULTS. BANKTEST time window is supposed to be used when testing scripts, routing and storage, at last nobody wants to send test messages to actual partners. On the other hand, while testing has finished, a rule can be established and set to DEFAULTS in order to allow document flow and processing to happen automatically.

Future work of the rule object should include a activation node, where rule can either be active or suspended, for example, to not create confusion inside own applications.

### 5.4.4 Messages

The Home component serves as the central dashboard where all messages in the system are displayed. It provides a clear and organized overview of incoming, outgoing, and processed messages, making it easy for users to monitor activity at a glance.

**Active Messages** Event Generator

Workflow ID	Sender	Object Type	Receiver	Step	Status	Assignee
5146ceea-3bc0-4ee2-bca7-74a40bfd659c	WARNING	ERROR	WARNING	WARNING	ERROR	actis
fdca4e56-8ed4-4163-9061-da3b2b52b5af	NissanMX	830	ToyotaSV	SEND	READY	actis

Resume  
Restart  
Assign  
Delete

Figure 5.21: Processing a message

All of these messages are ordered descending by the date they are created, each of them provides relevant information, as Sender, Object Type and Receiver, as well as status and assignee. A message of WARNING Sender and Receiver is a message which was received but a rule or certificate was not defined, hence it appears but it is not processable. All of the messages can be selected and RESUMED, RESTARTED, ASSIGNED or DELETED. All of them can be done in bulk, so multiple messages can be selected at once. Resuming a message will process its next step. Restarting a message will create a new message with identical settings in a new message. A message can be assigned to only one user, all of unassigned messages are assigned by actis(admin) user.

Filters are present, as many messages will be present usually, so filters can be very helpful in these situations. They allow users to narrow down the list of messages based on specific criteria such as sender, receiver, status, date, or message type. This makes it easier to find relevant messages quickly without having to scroll through long lists. Filters enhance usability and efficiency, especially in high-traffic systems, by helping users focus on what matters most and identify issues or track specific interactions with minimal effort.

### 5.4.5 Message Generation

Message generation in the system is driven by a combination of a rule and a file. The file typically contains the raw input data—such as structured content in any format — while the rule defines how that data should be interpreted, transformed, and structured into a valid message format (e.g., EDI, XML, or a custom layout). The rule may include mappings, validations, and logic to ensure that only the necessary information is extracted and arranged correctly according to the required standard.

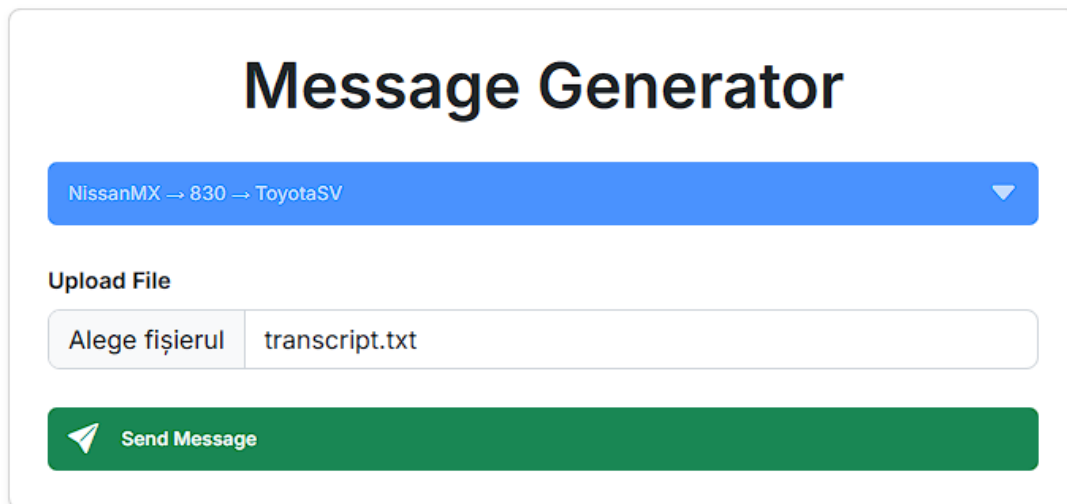
The image shows a web interface titled "Message Generator". At the top, there is a blue dropdown menu with the text "NissanMX → 830 → ToyotaSV" and a downward arrow. Below this, there is a section labeled "Upload File". Underneath, there is a light gray input field with the text "Alege fișierul" and "transcript.txt". At the bottom, there is a green button with a white paper plane icon and the text "Send Message".

Figure 5.22: Generating a message

### 5.4.6 Message Processing

Message processing in this system is designed to handle messages through a series of sequential steps, where each step performs a specific task—such as validation, transformation, routing, or logging. As the message moves through each step, the system captures and records the outcome, allowing you to see exactly how the message was handled at every stage. This approach makes it easier to track progress, identify errors early, and understand how data changes over time. By breaking processing into clear, manageable steps with visible results, it enhances transparency and helps ensure that messages are processed accurately and reliably.

NissanMX – 830 – ToyotaSV			
Step Name	Started Time	Ended Time	Result
<input type="text" value="Step Name"/>			<input type="text" value="Result"/>
COPY	Jun 10, 2025, 10:43:25 PM	Jun 10, 2025, 10:43:29 PM	✓ OK
SHELL	Jun 10, 2025, 10:43:29 PM	Jun 10, 2025, 10:43:33 PM	✓ OK
SEND	Jun 10, 2025, 10:43:33 PM		▶ READY
REMOVE			▶ READY

COPY ✕
SHELL ✕
SEND ✕

### SEND

```

ISA00      00      ZZNissanMXZZToyotaSV2405201435U004010000000010T:~
GSP0SENDERCODERECEIVERCODE2024052014351X004010~
ST8300001~
BFR00NE1234567892024052020240620BPDL~
SE50001~
GE11~
IEA1000000001~

```

Figure 5.23: Processing a message

Each message will contain a screen as the one above, showcasing at first each step, the time the step was processed and the result of each processing. In this example, the steps are COPY, SHELL, SEND, REMOVE, in this order, which it matters. First to copy the message in the archive and to create a relevant filename which can be found later, to debug or to re-process or whatever the user wants to do. After, the SHELL step, applies a SHELL script, in this case, it removes all '\*' characters in the file, the scripting is relative to what the user wants, and what the receiving party needs. After, a SEND step is being instantiated in order to SEND the converted message to the receiving party using the message text, mentioned in 5.4.3.

### 5.4.7 Archive

The archive acts as a central storage area where important system assets such as files, certificates, scripts, and other resources are securely saved and managed. It ensures that all components used in workflows—like input data files, communication certificates, and automation scripts—are organized, versioned, and easily retrievable when needed.

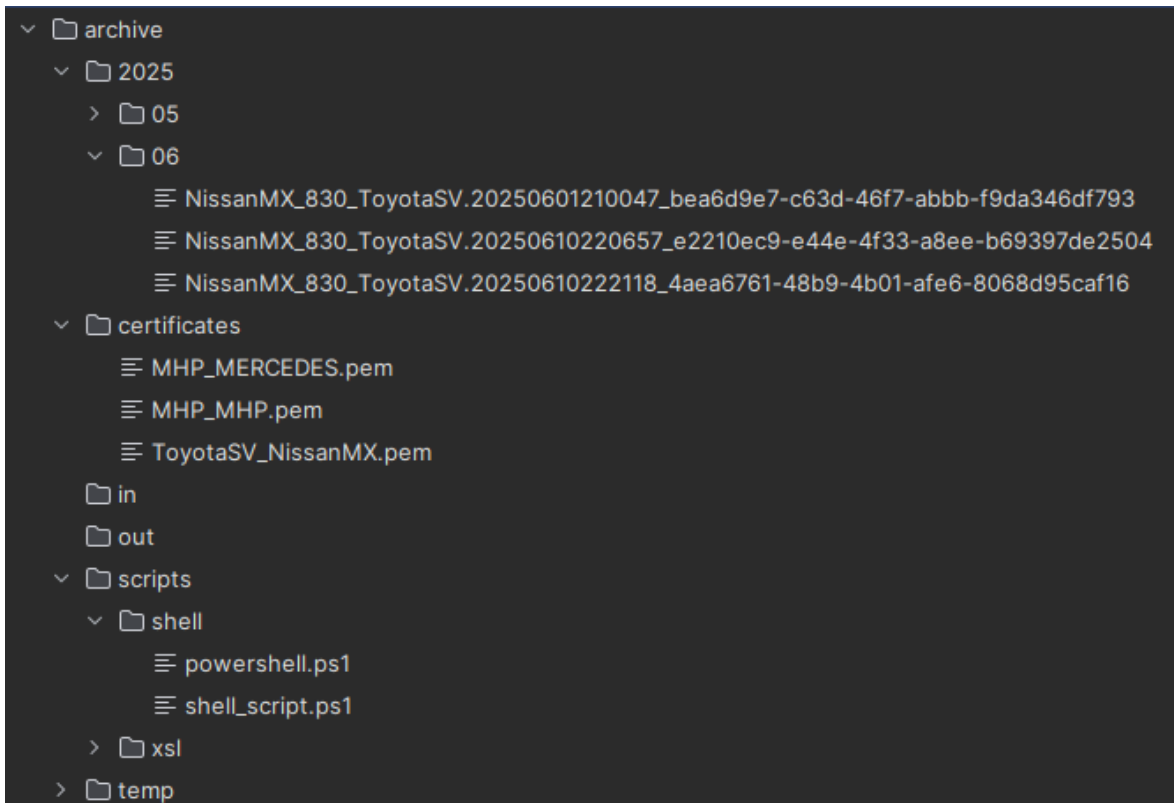


Figure 5.24: Processing a message

An archive is essential as files are stored having a significant name. The name is created as follows:

**Sender\_Object Type\_Receiver.Date Time\_Workflow Id**

The year and month directories are created automatically based on the date that the message is processed. Additionally, after a SHELL script is executed and a file is saved the filename is saved as follows (TF):

**Sender\_Object Type\_ReceiverTF.Date Time\_Workflow Id**

All the certificates and scripts are stored in the archive, the temp folder acts as a temporary folder, files after not being copied in the COPY step, are being sent first to the temp folder using the original filename, and are after renamed to the corresponding rule.

IN and OUT folders represent the services mentioned in 5.2.4.

By storing everything in one place, the archive simplifies management, enhances traceability, and supports reusability across different workflows or partners. It also plays a critical role in maintaining system integrity by keeping a record of all assets involved in message processing and integration tasks.

# Chapter 6

## RESULTS & PROSPECTS

In this chapter I will focus on the future work the application needs in order to be improved and potentially reach other competitors paid applications, additionally I will provide some challenges I faced during the implementation and what major set-backs I had during this time.

### 6.1 Future Work

An application can only be updated as much as the users need it, I believe that the application can be complete when users can add their own touch to the application.

#### 6.1.1 XML Conversion

In an ideal world, a user can define itself a set of rules where each message can be intercepted and transformed, not having only 2 standards to choose from, to create their own standards, to use company-specific standards, hence a XML conversion program could be the key for this. For example, you have an input message, where there are no standards defined yet, let's say it is a kind of programming language, and the user sets it's own kind of rules, a format description, in order to describe it's own set of messages. At the end of the day, this is how the most popular standards we're created. Let the user create this format description which can be applied to their type of message, resulting into a XML file, this being followed by a XSLT script, to map in whatever way the user wants it, and ultimately having a 'output' format description which reverts the file to another standard or format. The flow of the XML conversion should look something like this:



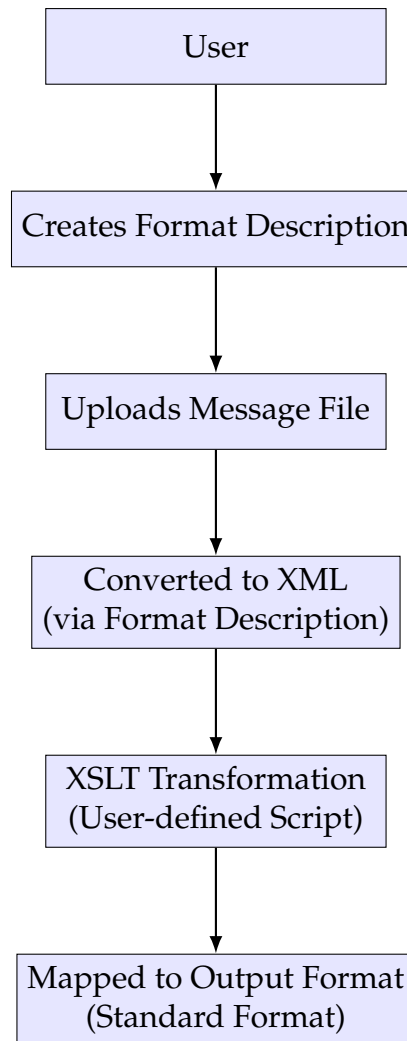


Figure 6.1: Workflow showing how a user defines a custom format description to convert a message file into XML, transform it using an XSLT script, and map the output back to a desired standard format.

### 6.1.2 User-Defined Workflows

Whilst this application works well with small to medium sized businesses, in order to make room for larger scale operations, the application itself can be integrated together with the server's operating system. Additionally while browsing through messages, the user's can use the system's operating system to connect and process messages there. One addition to the application can be to create user-defined workflows, using system commands. Instead of having only COPY, SEND, CONVERT, REMOVE, the user can define it's own steps using the server's command system. For example, having custom made workflows, can limit the work needed for automating them. This enhancement would not only increase operational efficiency by reducing manual intervention but also provide administrators with enhanced flexibility to customize processes tailored to their specific business needs. Ultimately,

enabling custom workflows to communicate with the OS would transform the application into a powerful orchestration tool, capable of managing both high-level business logic and critical infrastructure tasks in a unified and secure manner.

## **6.2 Conclusions**

The application demonstrates a well-structured and modular design that effectively addresses the needs of user management, configuration, secure access, and seamless backend communication. By organizing the system into clear modules such as Configuration, Login, Service, and Home, it ensures maintainability and scalability while promoting separation of concerns. The incorporation of user-defined format descriptions and flexible data transformation pipelines, including XML conversion and XSLT-based mapping, showcases the application's adaptability to diverse data standards and business requirements. Overall, this approach not only enhances the user experience by providing customizable workflows but also establishes a robust foundation for extending functionality in the future, making the application a versatile tool in any enterprise environment.

# Bibliography

- [Coa88] Phil Coathup. Electronic data interchange. *ITNOW*, 30(2):15–17, 1988.
- [CSP12] Jess Chadwick, Todd Snyder, and Hrusikesh Panda. *Programming ASP. NET MVC 4: Developing Real-World Web Applications with ASP. NET MVC*. " O'Reilly Media, Inc.", 2012.
- [Emm90] MA Emmelhainz. Electronic data interchange: A total management view, 1990.
- [GS13] Brad Green and Shyam Seshadri. *AngularJS*. " O'Reilly Media, Inc.", 2013.
- [GSWI95] Ian Graham, Graham Spinardi, Robin Williams, and Juliet Iwebster. The dynamics of edi standards development. *Technology Analysis & Strategic Management*, 7(1):3–20, 1995.
- [HF89] Ned C Hill and Daniel M Ferguson. Electronic data interchange: a definition and perspective. In *EDI Forum: The Journal of Electronic Data Interchange*, pages 5–12, 1989.
- [IBD95] Charalambos L Iacovou, Izak Benbasat, and Albert S Dexter. Electronic data interchange and small organizations: Adoption and impact of technology. *MIS quarterly*, pages 465–485, 1995.
- [Jan11] G Janssens. Electronic data interchange: from its birth to its new role in logistics information systems. *International Journal of Information Technology and Systems*, 3:45–56, 2011.
- [Lew98] Scott M Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys (CSUR)*, 30(1):3–27, 1998.
- [MF18] Anton Moiseev and Yakov Fain. *Angular Development with TypeScript*. Simon and Schuster, 2018.

- [O'N08] Elizabeth J O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356, 2008.
- [She23] Suman Shekhar. Framework for strategic implementation of sap-integrated distributed order management systems for enhanced supply chain coordination and efficiency. *Tensorgate Journal of Sustainable Technology and Infrastructure for Developing Countries*, 6(2):23–40, 2023.
- [SS92] Paula MC Swatman and Paul A Swatman. Edi system integration: A definition and literature survey. *The Information Society*, 8(3):169–205, 1992.
- [SSF94] Paula MC Swatman, Paul A Swatman, and Danielle C Fowler. A model of edi integration and strategic business reengineering. *The Journal of Strategic Information Systems*, 3(1):41–60, 1994.
- [Thu20] Sai Kumar Reddy Thumburu. The role of middleware in modern edi solutions. *Advances in Computer Sciences*, 3(1), 2020.
- [TL03] Thuan L Thai and Hoang Lam. . *NET framework essentials*. " O'Reilly Media, Inc.", 2003.