OXMES

- Open Extensible Manufacturing Execution System -

Project Report

Group deis106f18

Aalborg University
Department of Computer Science
Embedded Software Systems

Department of Computer Science
Aalborg University
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
OXMES - Open Extensible Manufacturing Execution System

Theme:
Distributed, Embedded and Intelligent Systems

Project Period:
Spring Semester 2018

Project Group:
deis106f18

Participants:
Anders Normann Poulsen
Gabriel Vasluianu

Supervisor:
Ulrik Nyman

Copies: 1

Page Numbers: 145

Date of Completion:
15th June 2018

Abstract:

The advent of Industry 4.0 encourages many researchers to study and conduct experiments in this field of work. Typical experiment scenarios include robots, control systems and product mockups. For better understanding of Modular Manufacturing Systems, commercial production modules are used, which come with their own proprietary control software. This is usually rigid, with a limited User Interface where things can be unnecessarily complicated. For this reason, better solutions are needed.
This project aims at implementing a Manufacturing Execution System, to be used for controlling the modular production line at the Department of Materials and Production of Aalborg University. It addresses the needs of the department for conducting their research. The project will be released as open source software. Furthermore, the methods and technologies used for its implementation, have been selected to take into account the lifetime of the project. This means that it should be easy for other developers to take on and continue, adapt or add features to the software.

Preface

This report is the Master thesis of the deis106f18 group following the Embedded Software Systems programme at Aalborg University.

We are thankful to Ulrik Nyman, our project supervisor, for great advice, flexibility and guidance throughout the last two semesters. We are grateful to Casper Schou from the Department of Materials and Production for the discussions and feedback related to the project. We would also like to thank Chen Li from the same department for the feedback offered on the API documentation.

Aalborg University, 15th June 2018

Anders Normann Poulsen
<apouls16@student.aau.dk>

Gabriel Vasluianu
<gvaslu16@student.aau.dk>

Reading Guide

In this report, references are written using the ACM in-text citation style. Sources are listed alphabetically in the bibliography. Figures, Tables, Equations, and Listings (code snippets) are numbered after the chapter they are in, for example, the first Figure in Chapter 2 is called 2.1, the next 2.2 etc. All illustrations and figures are made by the group members unless otherwise stated. All code can be found in the zip archive electronically uploaded with the report.
A prerequisite for reading this report is basic knowledge in the Computer Science field.

Glossary

API	Application Programming Interface
ACM	Association for Computing Machinery
CLI	Command-line interface
CLR	Common Language Runtime
CP	Cyber Physical
DDL	Data Description Language
EER	Enhanced entity-relationship
EF	Entity Framework
IIS	Internet Information Services
JSON	JavaScript Object Notation
MES	Manufacturing Execution System
MP	Department of Materials and Production
MMS	Modular Manufacturing Systems
MVC	Model-View-Controller
OPC UA	Open Platform Communication Unified Architecture
ORM	Object-relational mapping
PLC	Programmable Logic Controller
RFID	Radio-frequency Identification
UI	User Interface
UUID	Universally unique identifier
WPF	Windows Presentation Foundation
XML	Extensible Markup Language

Contents

CHAPTER 1

Summary

This report *"OXMES - Open Extensible Manufacturing Execution System"*, is the written result of the master thesis project for the Embedded Software Systems programme. The overall project comprises of two parts. In the first one, a specific problem related to Industry 4.0 has been investigated, and a system design has been proposed. The report for the first part can be found under the name of *"Extensible Manufacturing Execution System - Designing a modular, extensible MES for dependability and research"*. The second part of the project is the subject of this current report.

In this report we investigate the implementation of a Manufacturing Execution System (MES), to be used together with a Modular Manufacturing System. While the text contains concepts from the manufacturing and production fields, the work is mainly described from a computer science point of view. The report is made up of eight chapters, in which the premises of the project are being set, a problem statement is formulated, the system is further designed, and its implementation is described. Lastly, the work is evaluated and concluded.

The MES is implemented using a micro services architecture. The ASP.NET Core framework has been used to facilitate this. The implementation of three of the core services of the MES will be described in detail. Throughout the report there will be references and comparisons with the current MES used at the Department of Materials and Production. In the final chapters, the work is evaluated in terms of usability and possibility of further development. Known issues of the system, suggestions for features and possible improvements will be covered in the conclusion.

CHAPTER 2

Introduction

Industry 4.0 is a term used increasingly in the engineering and business world. The reason why this concept is popular, is because it promises to improve productivity, increase efficiency and lower costs for industrial applications. The way this is achieved, is by slightly moving away from the concept of *an automated manufacturing facility where humans have the task of supervising various processes*, to an era where *machines will manufacture products, while being able to reconfigure themselves, diagnose errors and provide insightful data about processes being run*.

Now, in order to know what practices to use in real-world applications, engineers and scientists have been conducting research, in different fields related to Industry 4.0. The Department of Materials and Production (MP) at Aalborg University is currently, among other things, focusing on controlling a Modular Manufacturing System (MMS). This entails having a control program, typically called a Manufacturing Execution System to monitor the actions of the manufacturing modules, to reconfigure these or to initiate orders of products to be built. The features of a MES facilitate experiments, demonstrations and student projects. This means that a MES has to be a versatile piece of software, preferably open-source, which researchers and students can easily work with, while being able to change different production parameters.

We have investigated the way the current MES works, and what possible improvements can be done to it. This has been done by first understanding how the MMS facility works, and what the role of the MES is, when controlling it. Then, a list of requirements has been compiled, based on numerous discussions with the users of the MES. These requirements have been categorised and prioritised. In the end, an architecture for a new MES has been designed, under the name of EMES (Ex-

tensible Manufacturing Execution System). All these activities, together with their results will be covered in the next chapter, which is based on our previous report [10].

In the meanwhile, we decided to change the name of the project to OXMES (Open Extensible Manufacturing Execution System), which is going to be used throughout this documentation.

CHAPTER 3

Previous work

The project we have been working on is divided into two parts. This chapter will mainly focus on the work done during the first part. The information presented here is brief; for more details, it is a good idea to refer to our previous report [10].

3.1 Mission statement

The first part of the project (9^{th} semester) was spent getting familiar with Industry 4.0 concepts, as well as studying the state of the art of this field. At the same time, we analysed the different aspects of the Modular Manufacturing System (MMS) used by the Department of Materials and Production (MP). The results of this first part of the project will be discussed in Section 3.2.

The purpose of this report is to give a detailed description of the work done during the second part (10^{th} semester) of the project. Section 3.3 contains a brief plan of the project's activities, while the following chapters go more in-depth into the implementation and evaluation of OXMES.

3.2 A new MES

The MMS at the Department of Materials and Production is a Festo CP (Cyber Physical) system. This is facilitating student projects, demonstrations and research. Figure 3.1 shows the arrangement of the Festo modules after the system has been reconfigured in the Spring of 2018. The reconfiguration activity had the goal of changing the placement of the Festo modules to show their versatility. First, the topology was changed in a tool called Experior, which took 3 minutes. The rearrangement and reconnection of the physical modules took 1 hour and 40 minutes.

Figure 3.1: Festo CP MMS at Department of Materials and Production

The intervals of time in which these two actions were executed come to show why it is desirable to use such a system in a fast-paced world.

A simple overview of the Festo CP reveals that the system consists of:

- The *modules* doing work on the products and transporting them on the conveyor belts
- The products being carried on special *carriers*, which are uniquely identified using RFID tags
- The *MES* running on a consumer grade computer

The PLCs inside each module wait for products on which they need to conduct operations. When a product arrives at a module, the PLC will check with the MES for product specific information (what needs to be done, and how).
Based on the observations of the system, and feedback received from MP, the group compiled a list of requirements for a better MES. This long list can be condensed in a few aspects that need to be worked on:

- Topology description
- Order handling
- Modular software architecture

This list of requirements hinted at building a new MES, to accommodate the needs of the MP department. Furthermore, building on top of the existing MES is not possible, as this is a closed-sourced project, developed by Festo. Having this information in mind, we decided to design a new MES based on *a micro service*

architecture. The result is represented by Figure 3.2.

The User Gateway is an endpoint for user interfaces to connect to. All the services of the system are connected to the User Gateway. Nine services have been defined, which achieve the different tasks of the MES. The MES is controlling and communicating with the low level hardware (the PLCs in the modules) through the PLC Gateway.

There are nine services that will take care of tasks in the work flow of manufacturing a product:

- **Topology Service**. Will make it possible to configure the physical arrangement of the modules.
- **Work Plan Service**. Keeps track of work plans. A work plan is a sequence of operations needed to manufacture a product.
- **Order Service**. Responsible for keeping track of product orders.
- **Order Scheduling Service**. Schedules the orders.
- **Operation Scheduling Service**. Schedules the operations in work plans, with regards to the topology of the system.
- **User Management Service**. Used to authenticate users.
- **Topology Discovery Service**. Used to automatically discover the topology of the system.
- **Watchdog Service**. Keeps track of the hardware status.
- **Analytics Service**. Provides metrics about the system.

OXMES 's main functionalities are based on the requirements discussed with and agreed upon with MP. There still are some minor concerns, for example, how does one develop a *good* User Management Service. Another thing to remember is not to make the services too dependent on the specific implementation of each other. Nevertheless, it should be easy to extend the system after it is handed in, thus the group should also focus on making decisions that facilitate this (programming language, documentation, application framework, etc.)

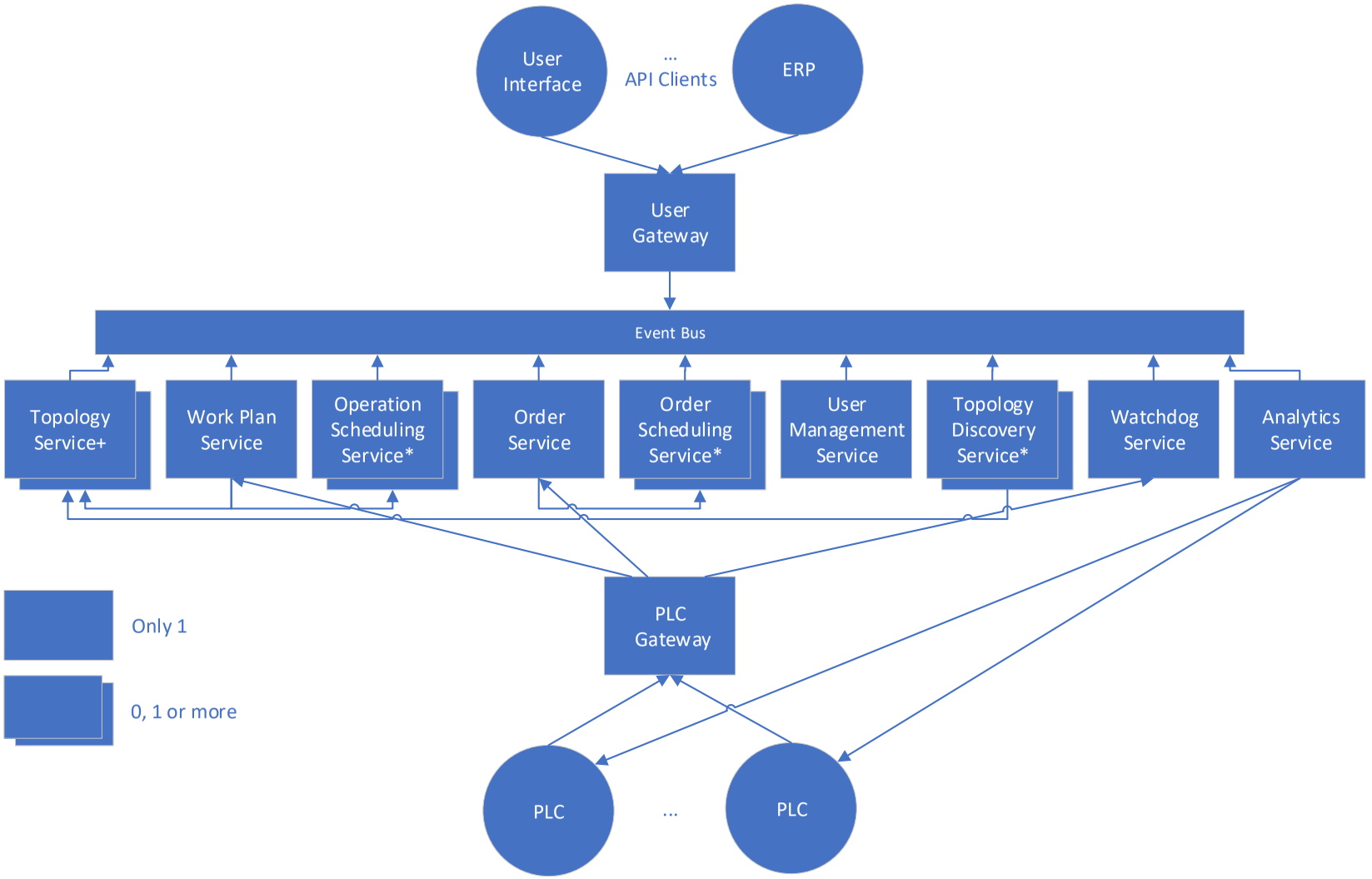


Figure 3.2: System architecture. Some services are optional, and some services may have multiple implementations running concurrently. A plus (+) illustrate that the service is required. A star (*) illustrate that the service is optional.

3.3 Development plan

The plan for the second part of this project is to implement OXMES while looking at the available technologies on which this can be built. This can be summarised as:

- Review the requirement list and prioritize this in terms of usability and importance for the overall system
- Find a suitable language to write the software, a framework on which to build the micro service architecture and a protocol to communicate with low level hardware
- Write the software for OXMES, model its databases and design its API
- Evaluate OXMES in terms of usability and extensibility

CHAPTER 4

Problem statement

This chapter describes the goal of this project in terms of: its evolution from previous work, milestones for this semester, and the knowledge gained while working on it.

Considering that we already decided to implement the architecture previously designed for OXMES, the goal we are mainly interested in, is related to the handover of this project. This can be formulated as:

> *How can we facilitate that the system being built will live on?*

Based on this goal, we can relate to a couple of other relevant aspects, which will be discussed, for the rest of report:

> *Have we followed the initial system design and user requirements?*
>
> *Do we comply with the user requirements?*
>
> *How will the end result be evaluated?*
>
> *What is the conclusion of this project and what remarks are there for future work?*

In a more objective manner, the obstacles that lie ahead are related to choosing the technologies to be used when implementing OXMES, together with proper development techniques for the software. The following chapter consists of a detailed overview of the decisions taken before actually starting to write code.

CHAPTER 5

System design

In this chapter, we will dive into analysing how the implementation of the different parts of the system should be prioritised. Following this, we will discuss how to implement the different features of the system, by looking at the programming language, frameworks and protocols to use.

5.1 Prioritisation

In this section, we establish a prioritised list of features of the system, and by extension therefore, which services from the micro service reference architecture to implement [10]. As this is just a half-year project, with only two people working on it, it is clear that covering the entire feature set of the reference architecture will be near impossible. Therefore, it is important to understand early, which features are necessary to operate the conveyors, issue orders and actually observe them being manufactured, such that the product of this project can adequately entice the Department of Materials and Production (MP) and/or the Department of Computer Science of Aalborg University to continue the development of OXMES.

The initial requirement list from [10] has been considered. The user stories in Sections 5.1.1 and 5.1.2 have been written based on discussions with MP. This resulted in the prioritised list of services in Section 5.1.3, which can also be seen as the order in which the services of OXMES should be implemented.

5.1.1 User story 1: Creating product orders

As a Smart Production Lab user, I want to be able to create and overview product orders, so that I can get the Festo CP running.

This user story focuses on having a way to initiate product orders, to be executed by the physical modules. The goal of the user might be either to produce something, or test the system as a whole. Either way, the order concept facilitates coordination between the physical modules, since it contains information about what actions need to be done and when. The result is, being able to input new orders, see existing orders, and watch the physical system execute them.

5.1.2 User story 2: Configuring the topology of the system

As a Smart Production Lab user, I want to be able to easily change the topology of the system, so that I can move or add production modules.

This user story deals with the topology of the physical modules in the system. The Lab users often change it for various reasons: demonstration, testing, experimenting, etc. The goal is to have an easy way to tell the MES what the new topology is, so that communication with the modules and scheduling of operations are done accordingly. The fact that changing the topology has to be done in an easy and intuitive manner has been brought into discussion by MP several times. The result would be compared with the existing way of defining new topologies, and evaluated in terms of usability.

5.1.3 Prioritised list of services

The following list contains the services envisioned in the architecture of OXMES. The list is a reiteration of the nine services introduced in Section 3.2, and it contains further details about their implementation. The user stories played an important role in prioritising the elements in the list. The prioritisation was also done with regards to the core services of OXMES (*Order-, Topology-, Order scheduling-, Work plan-* and *Operation Scheduling services*), thus these essential services are placed higher. However, the list does not contain the foundation of OXMES (*User Client*, *User Gateway* and *PLC Gateway*). This means that a good amount of time will be reserved for implementing the foundation, before working on the services.

The services have been divided into three categories, in regards to the priority of their implementation. First, the high priority services:

1. **Order service** is used to handle the order abstraction, containing all the information necessary for the products to be manufactured.
2. **Topology service** is needed to map the location of the physical modules. Additionally, it will contain a database of their capabilities. Without this service, it would be very hard to execute orders, and the system would not be as *reconfigurable* as intended.

Medium priority services:

3. **Order scheduling** is done based on their priorities and requested start times. This can also be done in a FIFO (First-In, First-Out) manner.
4. **Work plan** is the abstraction of the operations needed to be done in order to manufacture a product.
5. **Operation scheduling** can be done with regards to individual orders and their respective work plans. A more advanced approach would be to combine operations from different work plans to achieve production efficiency.
6. **User management** is worth looking into. The idea is for the system to be able to authenticate users and provide access to the MES's functionality accordingly.

Low priority services:

7. **Topology discovery** is a neat addon to the topology service. However, with a good topology service, the discovery is more of a luxury to have in a MES. Also, the way the modules are discovered, is a whole different subject than the goal of this report.
8. **Watchdog** is a nice-to-have feature, but will most likely not be implemented. We will assume that everything is working.
9. **Analytics** are not really needed at the moment.

5.2 Operation-to-application Mapping

In the Festo MES, each operation in a work plan is mapped directly to the application that must execute it. This puts the responsibility operation-to-application mapping on the operator creating the work plan. This is also the biggest hindrance to exploiting any inherent parallelism in the MMS.
Instead of this approach, we propose moving the operation-to-application mapping responsibility to the MES. To do so, we extend the applications with a list of supported operations. Additionally, we propose to describe operational ranges for each operation supported by an application, to support more generalised applications.

5.3 Language Choice

The goal of this project is not only to provide a running product, but also to provide a base for further development. The importance of the choice of language is

therefore even greater; not only should there be good library and framework support for the language, but the language should also be approachable, especially for production engineering students and faculty, who presumably have less exposure to the various modern languages compared to computer science students, and who may not be able to adjust to languages that are drastically different from ones they have already learnt.
Other than being easy to approach, there should also be good tools to develop micro-services in the language. This includes comprehensive and well documented frameworks and libraries. Additionally, the language, frameworks and libraries, should also have good community support, to allow us to quickly overcome any edge-case shortcomings in the official documentation. Lastly, to shorten the development lead up time, the language should be familiar to us.

To understand the programming language proficiencies of the students in the Department of Materials and Production, we approached our contact at the department, and asked which language he would prefer for himself and his students. He reckons that C would be the easiest language. Now, while C is a great language for embedded software, for other high level projects like this, it falls short in its language features and library support. But while the chosen language probably should not be C, to make transition easier for production engineering students, it should probably be C-like and structured and imperative. To sum it up, the following list contains the important conditions for choosing the programming language to use:

- C-like
- relatively easy to use and/or learn
- relative knowledge of the language both in MP, but also among the members of this group
- availability of libraries and frameworks necessary for OXMES
- cross-platform

Based on these requirements, we considered the following languages: C++, Java, JavaScript, C# and Python.

C++ is the only language on the list that is entirely unmanaged. This naturally increases the chances of memory leaks, compared to the other languages. While the object oriented nature of C++ might make the project more maintainable than it would be in C, it still suffers from problems with cross-platform development, requiring some platform specific code for all supported platforms, and individual compilation for each targeted platform. This makes it less than ideal for the purpose of this project.

Java is industry approved as a cross-platform language for enterprise software. This results in excellent community support and a vast collection of libraries. With Java EE and the Java API for RESTful Web Services (JAX-RS) specification, there is support for easy development of RESTful web APIs.

JavaScript is an old client-side scripting language primarily used on the web to compliment HTML. Recently with Node.JS, there has been a trend to use JavaScript for server-side development. Choosing JavaScript and Node.JS also allows for using TypeScript, a Microsoft language which is developed as a superset of JavaScript to overcome some of its weaknesses, with the inclusion of classes, optional types, generics and namespaces. TypeScript can be transpiled to JavaScript.
Unfortunately, the world of Node.JS evolves quickly, and the favourite framework of the month with developers is ever changing. This could make maintenance difficult, as the framework might go out of fashion within the lifetime of the software.

C# was traditionally a Windows-first language, with first-party support for Windows only, and limited third-party support for Unix-like systems via the Mono project. However, in recent years, Microsoft has moved towards Unix-like systems with the cross-platform .NET Core runtime and .NET Standard language specification. While true cross-platform GUI applications are still missing, console applications are finally truly cross-platform, which is of course enough for a web API. With the first-party web framework ASP.NET Core, developing a web API in C# has become a simple task.

Python is a language that facilitates quick prototyping, and focuses on code readability. Being a general-purpose language, it can be used for development of both small and large projects and it is supported on most modern operating systems. It features a lot of web frameworks, both low-level and high-level frameworks, so research of these is needed in order to make a choice for which one to use. Among the most popular and well maintained frameworks are: Flask and Django. Django in particular is well documented, includes tools that can be useful in a variety of projects and focuses on rapid development.

5.3.1 Decision

In the end, we decided to continue with C#. It has serious, first party support and documentation for good frameworks like ASP.NET Core and Entity Framework Core, which work across a host of platforms. With the exception of C++, it is also the language that is most similar to C. Additionally, MP has a researcher with a computer science background who is accustomed with C#. Lastly, this is the language that we have most collective experience with working in, which should make it easier to start work on constructing OXMES.

5.4 Frameworks

To simplify the development of the micro services, all services are implemented with the same framework: ASP.NET Core.

5.4.1 ASP.NET Core

ASP.NET Core is a reimplementation of Microsoft's famous ASP.NET framework. Together with .NET Standard and .NET Core, it forms the backbone of Microsoft's new push in the open source environment. For the first time ever, this trifecta of .NET tools provide a true first-party cross-platform C#, and .NET platform. This allows us to fulfil our requirement to be cross-platform, thus making development and maintenance easier, by not requiring a single platform. Within ASP.NET Core, the Model-View-Controller (MVC) framework is used to realise REST APIs. With simple class and method attributes, the framework handles routing of requests to specific methods.

As the name suggests, the MVC is made up of three complementary components. *The model* is working with data in an application. In trivial scenarios, it should be able to read data from a database and send it to the *view* of the application. An example for a more complex task is having to read data from a database, perform changes in regards to the *controller's* requests, and write the data back to the database. *The view* is the User Interface, and is only responsible for displaying information. It is the task of the *controller* to handle the requests from users and interact with the *model* [8].

5.4.2 Entity Framework Core

Entity Framework is an object-relational mapping (ORM) framework. It maps a database with a model, and abstracts away the underlying database. This ensures consistency between the model and the database, and allows developers to work directly on the objects without worrying about how to commit the changes to the database.

Entity Framework was part of the .NET Framework until version 6.0. With the launch of .NET Core, Entity Framework had to be completely rewritten to be adapted for .NET Core, and received a new name: Entity Framework Core (EF Core), and the version numbering reset to 1.0. Unlike the versions before, EF Core is not included in the .NET Framework, but is instead pulled as an external package.

There are two approaches when using EF Core, *database first* and *code first*. *Database first*, can be applied if a database already exists, or if you find it easier to start development with a database. In this case, EF Core will use reverse engineering to create a model that fits the database. *Code first* lets the you create your model in code, and EF will then generate the database fitting it, either by producing the necessary Data Description Language (DDL) script, or automatically at runtime. When using *code first*, EF Core supports two ways of describing the database constraints of model properties: *Fluent API* and *Data Annotations* [9]. With *Fluent API* developers add constraints to properties with code in the `DbContext` classes. For example, specifying the primary key of a model can be done like:

```
1  class TopologyContext : DbContext
2  {
3      public DbSet<Topology> Topologies;
4
5      protected override void OnModelCreating(ModelBuilder
           modelBuilder)
6      {
7          modelBuilder.Entity<Topology>()
8              .HasKey(t => t.TopologyId);
9      }
10 }
```

Listing 5.1: Specifying a primary key in Fluent API

Data Annotations uses property attributes in the model to describe constraints. Specifying a primary key, like in the example above can be done as follows with *Data Annotations*:

```
1  class Topology
2  {
3      [Key]
4      public static int TopologyId { get; set; }
5  }
```

Listing 5.2: Specifying a primary key with Data Annotations

While the *Fluent API* and *Data Annotations* represent two different approaches to accomplish the same task, both can be used together; in fact, as of EF Core 2.0, *Data Annotations* still does not support the full range of possibilities that *Fluent API* does, and it is therefore often necessary to use *Fluent API* for at least a part of the database specification in code.
When using the *code first* approach, EF Core also support Migrations. Migrations adds database version information to the database, and takes snapshots of the

model, to support automatically migrating data in the database upon changes to the model.

For the development of the OXMES's services, the *code first* approach will be used with primarily *Data Annotations*. While we have the old MES's Microsoft Access database, we decided not to use it directly as our model (as in the *database first* approach). The reason for this, is that it would be more flexible for us to design new, separate databases for the services, while including new features, and still use the original MS Access database as inspiration. On top of this, it would be easier and more transparent to implement the necessary data relationships ourselves using the EF's *code first* approach.

5.5 Protocols

OXMES will have to control low-level hardware. This can be done by using a communication protocol: OPC UA.

5.5.1 OPC UA

OPC UA (Open Platform Communication Unified Architecture) is a specification describing a machine-to-machine communication protocol. Its predecessor (Classic OPC) was tied to Windows machines, due to the fact that it used COM/DCOM as its main communication technology. The newer OPC UA is designed to use web service technologies, thus making it possible to be used on different platforms.

OPC UA uses the client/server model. Typically, low-level hardware, such as a PLC acts as an OPC UA server. A different service on the client side takes requests from the client application (this can be a control system or a web platform).

Figure 5.1 shows the OPC UA architecture. Inside the Server application, real life objects (such as sensors, valves) are represented as Nodes, in the OPC UA address space. By reading or writing values to the Nodes, the PLC's state can be looked at, or the PLC can be instructed to run actions. The server application also provides a subscription mechanism, in the form of monitored items, that regularly interact with the clients. Clients have access to the nodes by using the different services defined in the specification (such as *Read* or *Write*) [1].

OPC UA was recommended as the protocol to be used for communicating with the PLCs, by MP. On top of this, it is seen as the key standard in this field of work. OPC Foundation has released their OPC UA reference implementation for .NET Standard (and thus for .NET Core) [2]. This means that it should be trivial

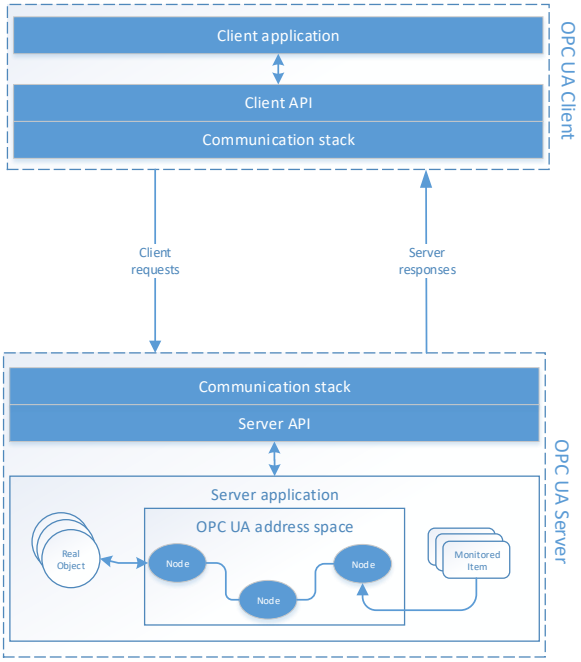


Figure 5.1: OPC UA (simplified) [3]

to embed the Client in our PLC Gateway service. As for the Server, which runs on the PLCs, it seems that Festo (the PLC manufacturer), has included the protocol in some of their products [11]. However, more research needs to be done in order to fully understand the compatibility of OPC UA with the production modules used at MP. The reason for not doing this, is that the PLC Gateway is not on the project backlog.

CHAPTER 6

System Implementation

This chapter will cover important details regarding the tools used for implementing OXMES. In the second part of the chapter, we will discuss interesting aspects of the work done on OXMES's services, and in the end we will evaluate our technical decisions.

6.1 Frameworks

The micro services are implemented using ASP.NET Core, and the data they work with is handled by Entity Framework Core. The following subsections contain details about these tools.

6.1.1 ASP.NET Core

ASP.NET Core with the built-in Model-View-Controller (MVC) framework and Kestrel web server makes development of RESTful web services easy.
Since we have no plan to implement a Web UI for any of the services in OXMES, the *view*-part of MVC will not be used. Instead, only the *model*- and *controller*-parts will be used. The framework includes routing middleware to automatically route requests to the relevant controller. The controller of an ASP.NET Core web server will serve the requests, and will generate appropriate responses to be sent back. In order for the framework to know exactly which controller to use when, each controller must use *decorators*. Take for example the decorator of a controller class as seen in Listing 6.1: any incoming requests to `/api/services` will be routed to the class `ServicesController` by the framework middleware. Methods in

this controller class can also be decorated with attributes of their own, to direct specific requests to them. One of the valid decorators is `[HttpGet]`. `[HttpGet]` will direct the middleware to route any GET requests to the decorated method. The decorator can also contain extra routing information, like `[HttpGet("id")]` which will direct the middleware to route any GET requests with a value appended on the route, to the decorated method. Regarding the methods we see in Listing 6.1: any HTTP GET request to `/api/services` will be executed in the method `ListServices()`, while any request like `/api/services/5` where 5 is an arbitrary ID, will be executed with the method `GetService(5)`. Similar decorators exist for the HTTP verbs DELETE, OPTION, PATCH, POST and PUT.
The use of these attributes make implementing RESTful APIs significantly easier.

```
1  [Route("api/services")]
2  public class ServicesController
3  {
4      [HttpGet]
5      public IActionResult ListServices()
6      {
7          // return a list of all services
8      }
9
10     [HttpGet("{id}")]
11     public IActionResult GetService(int id)
12     {
13         // return service with ID==id
14 }
```

Listing 6.1: Example of a class routing attribute: `[Route("api/services")]` and method routing attributes: `[HttpGet]` and `[HttpGet("id")]`

ASP.NET Core also includes Kestrel, an open source, cross-platform and lightweight web server. Using Kestrel allows for OXMES to run on the most common platforms, including Windows, macOS and Ubuntu. This allows for flexibility to develop and run OXMES on whichever platform is desired. As a lightweight web server, Kestrel does not support many complex features that web servers like Apache, IIS and nginx support. Therefore, the official documentation recommends against exposing Kestrel directly to the Internet. Instead Kestrel should be hidden behind a reverse proxy server like Apache, IIS or nginx.

6.1.2 Entity Framework Core

One of the things that is quickly learned when using Entity Framework Core (EF Core), is that there are often several ways to achieve the same result.

6.1.2.1 Database context

The entry point for EF Core is through a class extending the class `DbContext`. A model can be passed to EF Core through `DbSet<T>` properties in the `DbContext`, where `T` is a model class. This class, and all model classes it references will automatically be mapped to the database. It is also in `DbContext` that database constraints are defined with *Fluent API*.
Listing 6.2 shows an example of a `DbContext` class.

```
1  public class TopologyContext : DbContext
2  {
3      public DbSet<Models.Topology> Topologies { get; set; }
4      ⋮
5      protected override void OnModelCreating(ModelBuilder
           modelBuilder)
6      {
7          base.OnModelCreating(modelBuilder);
8
9          modelBuilder.Entity<TopologyModuleConveyor>()
10             .HasKey(tmc => new { tmc.ModuleConveyorId,
                   tmc.TopologyId });
11         ⋮
12     }
13 }
```

Listing 6.2: Parts of TopologyContext

6.1.2.2 Data relationships

The nature of a MES is to store data which is further associated with other data sets. For this reason, different types of relationships will have to be implemented between EF entities.

A *one-to-many* relationship is modelled as seen in Listing 6.3 and Listing 6.4. First, the parent entity defines a *collection* (it can also be an *enumerable*) of objects containing the child data. Then, in the model of the child, the parent property is included.

```
32 public virtual ICollection<OrderPosition>
       OrderPositions { get; set; } = new List<OrderPosition>();
```

Listing 6.3: Order model (parent), containing the OrderPosition property

```
23 public virtual Order Order { get; set; }
```

Listing 6.4: OrderPosition model (child), containing the Order property

By using Fluent API , the *one-to-many* relationship can be configured based on the properties in the two models, as seen in Listing 6.5:

```
24 protected override void OnModelCreating(ModelBuilder
       modelBuilder)
25 {
26     modelBuilder.Entity<Models.Order>()
27         .HasMany(op => op.OrderPositions)
28         .WithOne(o => o.Order)
29         .IsRequired();
30 }
```

Listing 6.5: OrderContext

In the case of *many-to-many* relationships, the child entity will also define a collection of objects containing the parent's properties. This is too confusing for EF Core; it can no longer automatically recognise the relationships. Instead, an extra associative class has to be created which binds a single parent to a single child. Both the parent class and child class then needs to contain a collection (or enumerable) of the associative class instead of direct references to one another. Lastly, the relationship must be configured using Fluent API. In Listing 6.6 we have an associative class called `TopologyModuleConveyor`, to store the keys of both a Topology and a Module Conveyor. Next, a *many-to-many* relationship is mapped between the `Topology` and `ModuleConveyor` entities.

```
protected override void OnModelCreating(ModelBuilder
    modelBuilder)
{
     modelBuilder.Entity<TopologyModuleConveyor>()
          .HasKey(tmc => new { tmc.ModuleConveyorId,
             tmc.TopologyId });

     modelBuilder.Entity<TopologyModuleConveyor>()
          .HasOne(tmc => tmc.Topology)
          .WithMany(t => t.TopologyModuleConveyors)
          .HasForeignKey(tmc => tmc.TopologyId);

     modelBuilder.Entity<TopologyModuleConveyor>()
          .HasOne(tmc => tmc.ModuleConveyor)
          .WithMany(mc => mc.Topologies)
          .HasForeignKey(tmc => tmc.ModuleConveyorId);
}
```

Listing 6.6: TopologyContext

It should be noted, that like many-to-many relationships, associative classes are also needed for one-to-one and one-to-many relationships where the parent and child are instances of the same class.

6.1.2.3 Foreign keys

EF Core is often able to figure out what the primary key of an entity should be. This is done by either looking for a property named using the class name appended with `Id`. For example, the property `OrderId` will automatically be used as primary key for the class `Order`, if no manual alternatives are provided. Primary keys can also be configured by adding the `[Key]` data annotation to any of the properties.

When working with data relationships, it is recommended to manually specify a foreign key to keep track of these relationships. This can be done by either including the `[ForeignKey]` data annotation in the model, or by calling `.HasForeignKey()` using Fluent API.

6.1.2.4 Serialisation

Serialisation of data needs to be considered when working with Web APIs. This is because it generally should be avoided to send too much data to a client at once.

This is both due to the bandwidth requirements, but more importantly to reduce unnecessary load on the database. Furthermore, sending implementation-specific data should be avoided, as it is not relevant for the client, and might even contain sensitive information.
By default, ASP.NET Core will serialise all public properties to JSON. To exclude a property, the `[JsonIgnore]` attribute can be used on the property.

6.2 Services

This section contains information about the services that have been implemented and experimented with as part of OXMES.

6.2.1 User Gateway

The User Gateway is the service positioned as gateway for the user clients into the micro service architecture. It is supposed to collect all API calls that the user clients might need into a single service. This allows the user clients to ignore the fact that the backend is a micro service architecture.
Since by the nature of the User Gateway it becomes critical that it is running for the whole system to be functional, and to reduce the total number of services, the User Gateway is also responsible for hosting service discovery, and hosting the Event Bus.

6.2.1.1 Service Discovery

Service discovery is implemented as a REST service of User Gateway. Services must announce their presence to the User Gateway with a HTTP POST using the `Service` model (see Section 6.2.1.3). Known services are stored in an in-memory database. This means that services must announce their presence every time they and the User Gateway start. Services may query the User Gateway for the identity of other services based on UUID, name or type. To see more about how services can do this, see Section 6.2.1.4.
When a service shuts down, it must either announce this with an HTTP DELETE, or if its connected to the Event Bus, with a Service Shutdown event.

6.2.1.2 Event Bus Server

The Event Bus is implemented with WebSockets. Along with a REST HTTP Server, the User Gateway implements a WebSocket server. Services should connect to the User Gateway with a WebSocket connection, and keep the connection open for the duration of its runtime. Using the Event Bus, services subscribe to events they

want to receive.
When a service generates an event, it must be sent to the User Gateway over WebSocket. The User Gateway will find all services that are subscribed to that type of event, and forward it to those services.

The implementation of the WebSocket server in ASP.NET Core has a quirk that should be noted, as it can possibly make the Event Bus Server extra vulnerable to denial of service (DOS) attacks. When a WebSocket clients connections to the WebSocket server, a new thread is created to service the connection. If this thread is closed, the WebSocket server implementation will automatically close the WebSocket connection. Therefore, to keep the WebSocket connection alive, the Event Bus Server is not allowed to let the request thread run to completion. Instead, each connection thread must enter a receive-loop, beginning with a blocking receive call. It is unknown whether the framework uses a thread pool, and will reuse threads while they are blocked.

6.2.1.3 Model

Though the User Gateway must be able to service many of the requests that other services handle, like getting a list of all topologies, these requests are mostly just forwarded to the responsible service. Therefore, the User Gateway does not need a model for the data sent in these requests.
The User Gateway does however have a model representation of a service.

Service This model represents services of the micro service architecture. The properties of the `Service` are as follows:

Name	Type	Description
UUID	Guid	A unique UUID assigned to the service by the User Gateway
Name	String	A unique name assigned to the specific implementation of the service
DisplayName	String	A pretty name of the service, used to display to the user
ServiceType	int	An integer denoting the service type
ServiceAddress	String	The URI to the service
IsActive	bool	Whether the service is active. Used when two services of the same type are present

6.2.1.4 Controller

User Gateway implements the following controllers: `ApplicationsController`, `ModuleConveyorsController`, `PingController`, `ResourcesController`, `ServicesController` and `TopologiesController`.
Among these controllers, the `Applications-`, `ModuleConveyors-`, `Resources-` and `TopologiesController` all just function as proxies to the Topology Service, and therefore their signature is identical to their corresponding equivalent in Topology Service, as seen in Section 6.2.3.3.

PingController exists to provide other services and clients with a way to identify whether the User Gateway is still alive. It supports just one request:

- Ping with message
  `GET api/ping/{message}`
  If User Gateway is ready responds with `200: {message}`, where `{message}` is an arbitrary string message passed as part of the route.

ServicesController provides an interface to services. This allows services to announce themselves and search for others. It supports the following requests:

- Get all registered services
  `GET api/services`
  Responds with status code `200` and JSON array of all registered services, if User Gateway is ready.
- Get service by UUID
  `GET api/services/{uuid}`, where `{uuid}` is a valid UUID
  Responds with:
  - `200`: And the `Service` with UUID = `{uuid}`.
  - `404`: If no services exist with the given UUID.
- Get active service by service type
  `GET api/services/type/{serviceType}`, where `{serviceType}` is an integer representing a service type
  Responds with:
  - `200`: And the `Service` with `ServiceType` = `{serviceType}`.
  - `204`: If no active services exist of the given type.
- Get by service name
  `GET api/services/name/{serviceName}`, where `{serviceName}` is a string
  Responds with:

  - `200`: And the `Service` with `Name` = `{serviceName}`.
  - `404`: If there is no service with the given name.

- Announce service
  `POST api/services`, with a `Service` object as request body
  Responds with:
  - `201`: And an identical `Service` object with the `UUID` field set.

- Update service
  `PUT api/services/{uuid}`, where `{uuid}` is the UUID of the `Service` to update, and with the updated `Service` object in the request body.
  Responds with:
  - `200`: If the service was updated.
  - `400`: If `{uuid}` does not match the UUID in the supplied `Service` object
  - `404`: If there is no service with the given UUID.

- Remove service
  `DELETE api/services/{uuid}`, where `{uuid}` is the UUID of the `Service` to remove.
  Responds with:
  - `200`: If the service was removed.
  - `404`: If there is no service with the given UUID.

To see more detailed documentation of the User Gateway API, see Appendix B.

6.2.2 User Management Service

Development of this service has been postponed indefinitely, due to complications with implementing it. ASP.NET Core provides an authentication and authorisation framework, and using this is strongly recommended by Microsoft, but how to use this framework in micro service architectures, where the user management service is *not* the gateway, is not documented well enough. Thus implementing it has proven difficult, and with limited time, it has been deemed better to focus on other absolutely essential services for the functionality of the MES.

6.2.3 Topology Service

The purpose of the Topology Service is to track the topology of the modules, resources and applications in the modular manufacturing system (MMS). This service also tracks the capabilities of applications, thus offloading operation-to-application mapping to the MES, rather than the operator.

6.2.3.1 Model

The model of the Topology Service consists primarily of the classes `Topology`, `ModuleConveyor`, `Resource`, `Application`, `ApplicationOperation`, `Range` and `RangeParameter`. Objects of these classes are available through the service controllers. Aside from these, are also the classes `TopologyModuleConveyor`, `ConveyorLink` and `ModuleConveyorConveyorLink`, which exist primarily for EF Core to correctly map the relationships of the other model classes. `TopologyModule- Conveyor`, `ConveyorLink` and `ModuleConveyorConveyorLink` are all serialised into other formats, ad thus not exposed to clients.
Figure 6.1 shows all the model classes and their relationships.

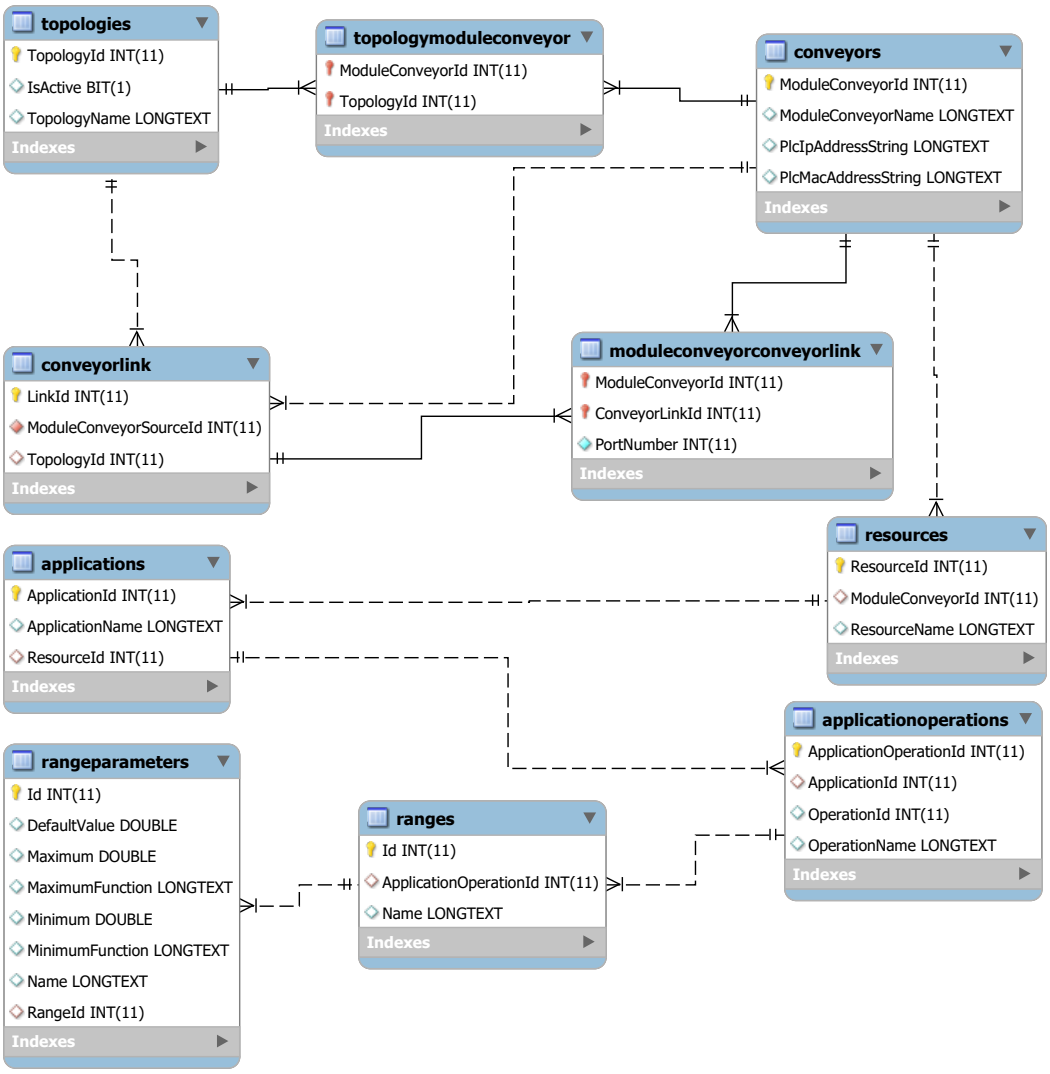


Figure 6.1: An EER Diagram of the model for the topology service

`Topology` represents a topology. The properties of `Topology` are as follows:

Name	Type
TopologyId	int
TopologyName	string
IsActive	bool
TopologyModuleConveyors	IEnumerable<TopologyModuleConveyor>
ConveyorLinks	IEnumerable<ConveyorLink>
ConveyorIds	int[]
ConveyorLinkIds	IDictionary<int, IDictionary<int, int»

The properties `TopologyModuleConveyors` and `ConveyorLinks` are used to help EF Core model the relations to ModuleConveyor and ConveyorLinks. When serialised, these turn into `ConveyorIds` and `ConveyorLinkIds` respectively. `ConveyorIds` is an array of the ID's of the module conveyors in the topology. `ConveyorLinkIds` denotes the links in a topology, in a more serialiser-friendly manner. The first integer in this construct denotes the ID of a module conveyor, the second integer denotes the port index of the link, and the last integer denotes the ID of the destination module conveyor.

`ModuleConveyor` represents a module conveyor. The properties of `Module-Conveyor` are as follows:

Name	Type
ModuleConveyorId	int
ModuleConveyorName	string
PlcMacAddressString	string
PlcMacAddress	PhysicalAddress
PlcIpAddressString	string
PlcIpAddress	IPAddress
Resource	Resource
ResourceId	int?
Topologies	IEnumerable<TopologyModuleConveyor>
IncomingConveyorLinks	IEnumerable<ModuleConveyorConveyorLink>
OutgoingConveyorLinks	IEnumerable<ConveyorLink>

The properties `PlcMacAddressString` and `PlcIpAddressString` are string representations of their `PhysicalAddress` and `IPAddress` counterparts. They exist to make serialisation easier, both to JSON, but also to provide EF Core with a format it can map to the database, as neither `PhysicalAddress` nor `IPAddress` are types EF Core supports.
The properties `Resource`, `Topologies`, `IncomingConveyorLinks` and

`OutgoingConveyorLinks` exist to help EF Core understand the relationships between the different types. To prevent circular references, and to reduce their size, these properties are not serialised and included in GET responses.

`TopologyModuleConveyor` exists to help EF Core create a many-to-many relationship between `Topology` and `ModuleConveyor`. The properties of this class are:

Name	Type
TopologyId	int
Topology	Topology
ModuleConveyorId	int
ModuleConveyor	ModuleConveyor

As EF Core does not support automatic discovery of many-to-many relationships, this class also utilises *Fluent API* to map to the database. The *Fluent API* for this class is:

```
26 protected override void OnModelCreating(ModelBuilder
       modelBuilder)
27 {
28     modelBuilder.Entity<TopologyModuleConveyor>()
29         .HasKey(tmc => new { tmc.ModuleConveyorId,
               tmc.TopologyId });
30
31     modelBuilder.Entity<TopologyModuleConveyor>()
32         .HasOne(tmc => tmc.Topology)
33         .WithMany(t => t.TopologyModuleConveyors)
34         .HasForeignKey(tmc => tmc.TopologyId);
35
36     modelBuilder.Entity<TopologyModuleConveyor>()
37         .HasOne(tmc => tmc.ModuleConveyor)
38         .WithMany(mc => mc.Topologies)
39         .HasForeignKey(tmc => tmc.ModuleConveyorId);
40 }
```

Listing 6.7: Fluent API code for TopologyModuleConveyor

`ConveyorLink` together with `ModuleConveyorConveyorLink`, represents a connection between two conveyors. The properties of this class are:

Name	Type
LinkId	int
ModuleConveyorSourceId	int
ModuleConveyorSource	ModuleConveyor
DestinationModuleConveyor-ConveyorLinks	ICollection<ModuleConveyor-ConveyorLink>

A link between two module conveyors is described by a triple of properties: a source module conveyor, the port on the source connected to the destination, and a destination module conveyor. The property `ModuleConveyorSource` denotes the source module conveyor. The port index, and the destination module conveyor appear in the `ModuleConveyorConveyorLink` class.

To hide additions to the model that exist only to please EF Core from clients, objects of this class are not directly serialised. To see the actual format used for serialisation, see the description of the `Topology` model.
The *Fluent API* used to define the relationships of this class is shown in Listing 6.8.

```
protected override void OnModelCreating(ModelBuilder
   modelBuilder)
{
    modelBuilder.Entity<ConveyorLink>()
             .HasOne(cl => cl.ModuleConveyorSource)
             .WithMany(mc => mc.OutgoingConveyorLinks);
}
```

Listing 6.8: Fluent API code for ConveyorLink

`ModuleConveyorConveyorLink` represents the relationship between `Module-Conveyor` and `ConveyorLink`. This class exists to help EF Core understand the many-to-many relationship between the two classes. The properties of this class are:

Name	Type
ModuleConveyorId	int
ModuleConveyor	ModuleConveyor
ConveyorLinkId	int
ConveyorLink	ConveyorLink
PortNumber	int

A link between two module conveyors is described by a triple of properties: a source module conveyor, the port on the source connected to the destination, and

a destination module conveyor. The property `ModuleConveyorSource` of `ConveyorLink` denotes the source module conveyor. `PortNumber` denotes the port index (0-based), and `ModuleConveyor` denotes the destination module conveyor.

To hide additions to the model that exist only to please EF Core from clients, objects of this class are not directly serialised. To see the actual format used for serialisation, see the description of the `Topology` model.
The *Fluent API* used to define the relationships of this class is shown in Listing 6.9.

```
protected override void OnModelCreating(ModelBuilder
   modelBuilder)
{
    modelBuilder.Entity<ModuleConveyorConveyorLink>()
        .HasKey(mccl => new { mccl.ModuleConveyorId,
           mccl.ConveyorLinkId });

    modelBuilder.Entity<ModuleConveyorConveyorLink>()
        .HasOne(mccl => mccl.ConveyorLink)
        .WithMany(cl =>
           cl.DestinationModuleConveyorConveyorLinks)
        .HasForeignKey(mccl => mccl.ConveyorLinkId);

    modelBuilder.Entity<ModuleConveyorConveyorLink>()
        .HasOne(mccl => mccl.ModuleConveyor)
        .WithMany(mc => mc.IncomingConveyorLinks)
        .HasForeignKey(mccl => mccl.ModuleConveyorId);
}
```

Listing 6.9: Fluent API code for ModuleConveyorConveyorLink

`Resource` represents a resource in a topology. At present, it has the following properties:

Name	Type
ResourceId	int
ResourceName	string
ModuleConveyorId	int?
ModuleConveyor	ModuleConveyor
Application	Application
ApplicationId	int?

The property `ModuleConveyor` denotes the module conveyor on which this resource is hosted. `Application` denotes the application associated with this resource. A resource can be an orphan; both ModuleConveyor and Application are optional.
Only the properties `ResourceId`, `ResourceName`, `ModuleConveyorId` and `ApplicationId` are serialised to JSON to be transmitted in HTTP requests and responses; `ModuleConveyor` and `Application` are not included, both to reduce the amount of data sent, and to prevent circular references.

`Application` represents an application attached to a resource. The properties of an application are:

Name	Type
ApplicationId	int
ApplicationName	string
ResourceId	int?
Resource	Resource
SupportedOperations	IEnumerable<ApplicationOperation>

The property `SupportedOperations` is a list of all operations, a given application supports.
The property `Resource` exists to help EF Core understand the relationship between Application and Resource. It is not serialised to JSON to be transmitted in HTTP requests and responses. This property is optional; an Application can exist without being attached to a Resource.

`ApplicationOperation` represents a single operation an application can perform. This class contains the following properties:

Name	Type
ApplicationOperationId	int?
OperationId	int?
OperationName	string
ApplicationId	int?
Application	Application
SupportedRanges	IEnumerable<Range>

Despite their similar names the two properties `ApplicationOperationId` and `OperationId` have very different purposes.
The property `ApplicationOperationId` is the primary key of this class, and is used only for identification of a particular ApplicationOperation. `OperationId`

on the other hand, denotes the ID of a particular operation. This ID should be identical to the ID of an operation in work plans to perform a certain operation. Despite the relationship between operation ID in ApplicationOperation and work plans, there is no directly link between them, and therefore no enforcement. The `OperationName` is an arbitrary human readable descriptor of the operation, but it is not enforced whether `OperationName` is identical between all Application-Operation with the same `OperationId`.

`SupportedRanges` is a list of all the supported n-dimensional operating ranges a particular application can perform this particular operation.

The property `Application` exists to help EF Core understand the relationship between Application and ApplicationOperation. It is not serialised to JSON to be transmitted in HTTP requests and responses.

`Range` represents an n-dimensional operating range of a particular operation for a particular application. It contains the following properties:

Name	Type
Id	int?
Name	string
ApplicationOperationId	int?
ApplicationOperation	ApplicationOperation
Parameters	IEnumerable<RangeParameter>

The property `Parameters` is a list of supported n dimensions in a range. A dimension does not have to denote a physical dimension, such as movement in X-, Y- or Z-space, but it can also denote an operation specific attribute, such as a specific drill bit on for drilling.

The property `ApplicationOperation` exists to help EF Core understand the relationship between Application and ApplicationOperation. It is not serialised to JSON to be transmitted in HTTP requests and responses.

`RangeParameter` represents a single dimension of a range. This class contains the following properties:

Name	Type
Id	int?
Name	string
RangeId	int?
Range	Range
Minimum	double?
Maximum	double?
MinimumFunction	string
MaximumFunction	string
DefaultValue	double?

The properties `Minimum` and `Maximum` denote the absolute minimum and maximum values supported in a dimension.

The property `DefaultValue` denotes the value to use for work plan operations if this particular dimension (RangeParameter) is not specified in the work plan. If this value is not set, this dimension becomes required; all work plan operations must specify this dimension to be supported in this range.

The properties `MinimumFunction` and `MaximumFunction` denotes mathematical functions describing the minimum and maximum. They exist to support applications with irregular operating ranges, where the operating range in a dimension is dependant on the location of one or more other dimensions. The format of these properties should be like `{X}**2`, where `X` is the name of another dimension (RangeParameter) in the same range.

`RangeParameter`s are identified only by the property `Name`; the property `Id` exists only for database purposes. Similarly, the property `Range` exists only to help EF Core.

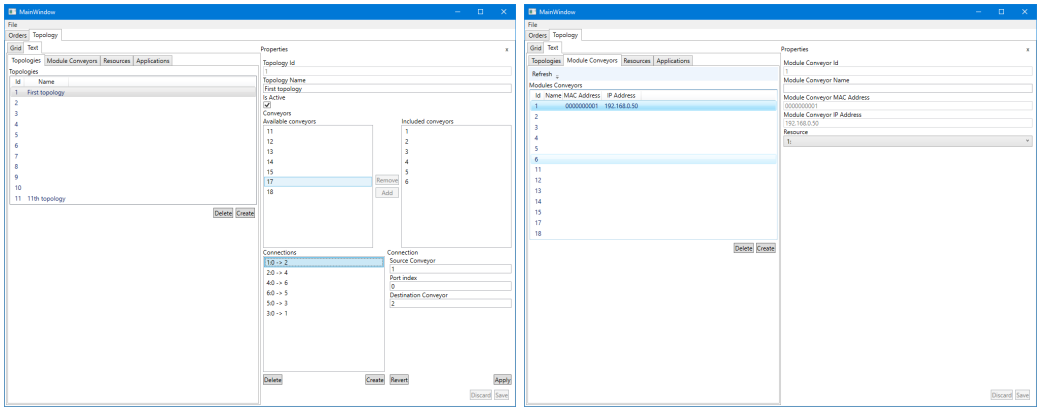
6.2.3.2 View

While the Topology Service does not have any built-in view, it can be visualised through the user client.
The user client contains a tab for all data related to topology. Within this tab are subtabs for Topologies, Module Conveyors, Resources and Applications. Each of these tabs contain a list of all the corresponding items in the database to the left, and when an item is selected, a properties panel will appear to the right. So the Topologies tab contains a list of all topologies in the database to the left, and when a topology from this list is selected, a panel with all properties of the topology will appear to the right.

All tabs for the Topology Service allow editing items and pushing those edits to the Topology Service. They also allow creating and deleting items.

Figure 6.2a (full size in Figure A.1) shows the Topologies tab. As well as the simple properties, like ID and name, the topologies properties panel contains both a list of all module conveyors included and not included in the topology. It also contains a list of all links in the topology, visualised in text with the format `{source module conveyor}:{port index} -> {destination module conveyor}`

Figure 6.2b (full size in Figure A.2) shows the Modules Conveyor tab. The properties panel shows the simple properties ID, name, IP- and MAC address, as well as a drop down list of all known resources, denoting which resource is associated with the module conveyor.

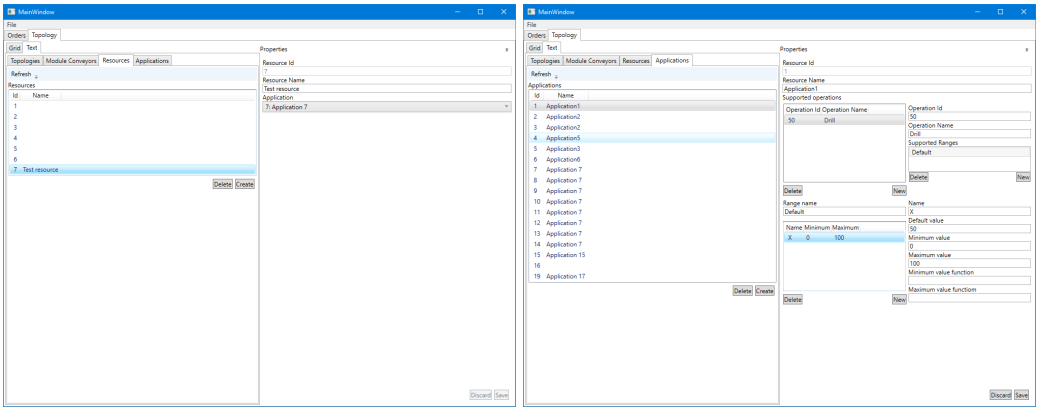


(a) Topologies tab **(b)** Module Conveyors tab

Figure 6.2: Screenshots of the user client

Figure 6.3a (full size in Figure A.3) shows the Resources tab. The properties panel shows the simple properties ID and name, as well as a drop down list of all known applications, denoting which resource is on top of the resource.

Figure 6.3b (full size in Figure A.4) shows the Applications tab. The properties panel shows the simple properties ID and name, as well as lists of all supported operations, ranges and dimensions (range parameters), and the properties for each of these; when an operation is selected, all properties of that operation are shown to the right; when a range is selected, all properties of that range are further shown below; and when a range parameter is selected, all properties of that range are further shown to the right.



(a) Resources tab **(b)** Applications tab

Figure 6.3: Screenshots of the user client

6.2.3.3 Controller

Topology Service implements the following controllers: `ApplicationsController`, `ModuleConveyorsController`, `ResourcesController` and `TopologiesController`.

ApplicationsController handles requests pertaining to applications. It supports the following requests:

- Get all applications
  `GET api/applications`
  Responds with status code `200` and a JSON array of all applications, if Topology Service is ready. The applications in the JSON array do not include nested objects.
- Get application by ID
  `GET api/applications/{id}`, where `{id}` is a valid ID
  Responds with:
  - `200`: And the `Application` with `ApplicationID` = `{id}`, including nested objects.
  - `404`: If no application exists with the given ID.
- Create application
  `POST api/applications`, with an `Application` object as request body
  Responds with:

  - `201`: And an identical `Application` object with the `ApplicationID` field set.

- Update application
  `PUT api/applications/{id}`, where `{id}` is the ID of the `Application` to update, and with the updated `Application` object in the request body.
  Responds with:
  - `200`: If the application was updated.
  - `400`: If `{id}` does not match the ID in the supplied `Application` object
  - `404`: If there is no application with the given ID.

- Remove application
  `DELETE api/applications/{id}`, where `{id}` is the ID of the `Application` to remove.
  Responds with:
  - `200`: If the application was removed. Response body contains a copy of the removed application.
  - `404`: If there is no application with the given ID.

ModuleConveyorsController handles requests pertaining to module conveyors. It supports the following requests:

- Get all module conveyors
  `GET api/moduleconveyors`
  Responds with status code `200` and JSON a array of all module conveyors, if Topology Service is ready. The module conveyors in the JSON array do not include information about resources.

- Get module conveyor by ID
  `GET api/moduleconveyors/{id}`, where `{id}` is a valid ID
  Responds with:
  - `200`: And the `ModuleConveyor` with `ModuleConveyorID = {id}`, including resource objects.
  - `404`: If no module conveyor exists with the given ID.

- Create module conveyor
  `POST api/moduleconveyors`, with a `ModuleConveyor` object as request body
  Responds with:

  - `201`: And an identical `ModuleConveyor` object with the `ModuleConveyorID` field set.

- Update module conveyor
  `PUT api/moduleconveyors/{id}`, where `{id}` is the ID of the `ModuleConveyor` to update, and with the updated `ModuleConveyor` object in the request body.
  Responds with:

  - `200`: If the module conveyor was updated.
  - `400`: If `{id}` does not match the ID in the supplied `ModuleConveyor` object
  - `404`: If there is no module conveyor with the given ID.

- Remove module conveyor
  `DELETE api/moduleconveyors/{id}`, where `{id}` is the ID of the `ModuleConveyor` to remove.
  Responds with:

  - `200`: If the module conveyor was removed. Response body contains a copy of the removed module conveyor.
  - `404`: If there is no module conveyor with the given ID.

ResourcesController handles requests pertaining to resources. It supports the following requests:

- Get all resources
  `GET api/resources`
  Responds with status code `200` and a JSON array of all resources, if Topology Service is ready. The resources in the JSON array do not include information about applications.

- Get resource by ID
  `GET api/resources/{id}`, where `{id}` is a valid ID
  Responds with:

  - `200`: And the `Resource` with `ResourceID = {id}`, including information about application.
  - `404`: If no resource exists with the given ID.

- Create resource
  `POST api/resources`, with a `Resource` object as request body
  Responds with:

  - `201`: And an identical `Resource` object with the `ResourceID` field set.

- Update resource
  `PUT api/resources/{id}`, where `{id}` is the ID of the `Resource` to update, and with the updated `Resource` object in the request body.
  Responds with:
  - `200`: If the resource was updated.
  - `400`: If `{id}` does not match the ID in the supplied `Resource` object
  - `404`: If there is no resource with the given ID.

- Remove resource
  `DELETE api/resources/{id}`, where `{id}` is the ID of the `Resource` to remove.
  Responds with:
  - `200`: If the resource was removed. Response body contains a copy of the removed resource.
  - `404`: If there is no resource with the given ID.

TopologiesController handles requests pertaining to topologies. It supports the following requests:

- Get all topologies
  `GET api/topologies`
  Responds with status code `200` and a JSON array of all topologies, if Topology Service is ready. The topologies in the JSON array do not include information about module conveyors and conveyor links.

- Get topology by ID
  `GET api/topologies/{id}`, where `{id}` is a valid ID
  Responds with:
  - `200`: And the `Topology` with `TopologyID` = `{id}`, including information about module conveyors and conveyor links.
  - `404`: If no topology exists with the given ID.

- Create topology
  `POST api/topologies`, with a `Topology` object as request body
  Responds with:
  - `201`: And an identical `Topology` object with the `TopologyID` field set.

- Update topology
  `PUT api/topologies/{id}`, where `{id}` is the ID of the `Topology` to update, and with the updated `Topology` object in the request body.
  Responds with:
  - `200`: If the topology was updated.
  - `400`: If `{id}` does not match the ID in the supplied `Topology` object
  - `404`: If there is no topology with the given ID.
- Remove topology
  `DELETE api/topologies/{id}`, where `{id}` is the ID of the `Topology` to remove.
  Responds with:
  - `200`: If the topology was removed. Response body contains a copy of the removed topology.
  - `404`: If there is no topology with the given ID.

For more detailed documentation of the API exposed by the controllers of Topology Service, see Appendix C.

6.2.4 Order Service

The purpose of the Order Service is to receive orders from one of the user clients, and store them. These will then be handled by another service, which will schedule them accordingly, and based on their work plans, instructions will be sent to the physical modules (the PLCs). Our vision is that these instructions will be routed through the PLC Gateway.

6.2.4.1 Model

The model of the Order Service consists of three classes `Order`, `OrderPosition` and `Product`. Objects of these classes are available through the service controllers. Figure 6.4 shows the model classes and their relationships.

`Order` represents an order created by a user. This can be seen as a group of multiple products that need to be produced. For this reason an Order is made up of one or multiple Order Positions, the latter representing the order-specific properties of individual products on the line.
This class contains the following properties:

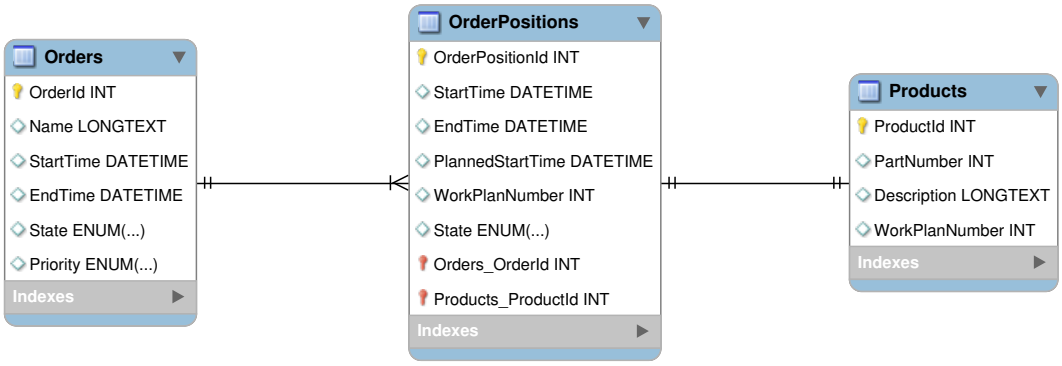


Figure 6.4: An EER Diagram of the models for the Order Service

Name	Type
OrderId	int
Name	string
StartTime	DateTime
EndTime	DateTime
PlannedStartTime	DateTime
State	Enum
Priority	Enum
OrderPositions	ICollection<OrderPosition>

Properties `StartTime`, `EndTime`, `PlannedStartTime` are an indication for the order progress while being handled. This data can be later used by an eventual Analytics Service to compile statistics about the efficiency of the system. Note that there is no `PlannedEndTime` property, as it would be hard to estimate this value, without knowledge about the scheduling of the orders. For the moment, it was decided to be more applicable to disregard such a property.
Properties `State` and `Priority` denote the state and the priority of an order. These are modelled as enums, as seen in Listings 6.10 and 6.11.
Lastly, `OrderPositions` exists to help EF Core understand the relationship between Order and OrderPosition.

```
1 public enum State
2 {
3         Created,
4         Started,
5         Completed
6 }
```

Listing 6.10: State enum

```
1 public enum Priority
2 {
3         Low,
4         Medium,
5         High
6 }
```

Listing 6.11: Priority enum

`OrderPosition` represents the abstraction of an individual product that is going to be manufactured, based on an existing order.
This class contains the following properties:

Name	Type
OrderPositionId	int
StartTime	DateTime
EndTime	DateTime
PlannedStartTime	DateTime
WorkPlanNumber	int
State	State (enum)
OrderId	int?
Order	Order
ProductId	int?
Product	Product

Order Position has a few similar properties with Order. It was discussed if these should be inherited from it, but the idea was not implemented due to the possibility of over-complicating the model. `WorkPlanNumber`, `Product` are properties that help identify the characteristics of a physical product.
`OrderId`, `Order`, `ProductId`, `Product` exist to help EF Core understand the relationship between Order Position and both Order and Product.

`Product` represents a product that the Festo MMS can build. Based on this model, a library of products can be added to the database, having each OrderPosition refer to these.
This class contains the following properties:

Name	Type
ProductId	int
PartNumber	int
Description	String
WorkPlanNumber	int
OrderPositionId	int?
OrderPosition	OrderPosition

`PartNumber` is a unique identifier for the products that can be manufactured. `WorkPlanNumber` is the same property that we have in the Order Position. It is possible that in the future this should only be part of one of these models.
Again, `OrderPositionId` and `OrderId` exist to help EF Core understand the relationship between Product and Order Position.

6.2.4.2 View

The plan is to have the Order Service viewed through the user client. This was not implemented yet. In Figure 6.5 (full size in Figure A.5), the envisioned view for the Order Service tab can be seen. For the moment, this was hard-coded in the user client, but can definitely be used as a template for implementing the functionality of the service.

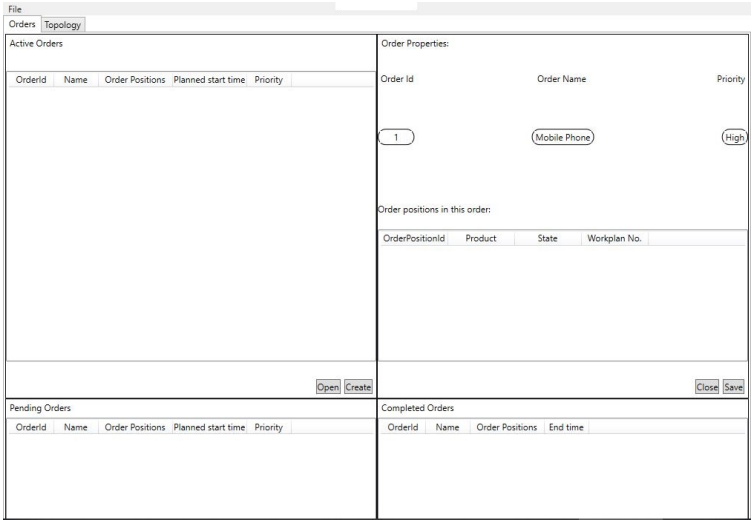


Figure 6.5: Screenshot of Order view

One of the important things that can be seen in the Order view is the separation of past, present and future orders (in the *Active-*, *Pending-* and *Completed Orders* panels). This is one of the requirements for this service.
Each of these panels will display orders based on the same model described in

Section 6.2.4.1. However, there is no need to display all the properties of an order. For example, in the *Completed Orders* panel, there is no need to show the state, priority or planned started time of an order, as we already know that the orders in this panel are completed.

6.2.4.3 Controller

The service has three controllers for each of the models in Section 6.2.4.1.

OrdersController handles requests pertaining to orders. It supports the following requests:

- Get all orders
  `GET api/Orders`
  Responds with status code `200` and a JSON array of all orders. The orders in the JSON array include nested objects.
- Get order by ID
  `GET api/Orders/{id}`, where `{id}` is a valid ID
  Responds with the `Order` with `OrderId` = `{id}`, including nested objects.
- Create order
  `POST api/Orders`, with an `Order` object as request body
  Responds with status code `201` and an identical `Order` object with the `OrderId` field set.
- Update order
  `PUT api/Orders/{id}`, where `{id}` is the Id of the `Order` to update, and with the updated `Order` object in the request body.
  Responds with `200` if the order was updated.
- Remove order
  `DELETE api/Orders/{id}`, where `{id}` is the ID of the `Order` to remove.
  Responds with status code `200` if the order was removed. Response body contains a copy of the removed order.

OrderPositionsController handles requests pertaining to order positions. It supports the following requests:

- Get all order positions
  `GET api/OrderPositions`
  Responds with status code `200` and a JSON array of all order positions. The orders in the JSON array include nested objects.

- Get order position by ID
  `GET api/OrderPositions/{id}`, where `{id}` is a valid ID
  Responds with the `OrderPosition` with `OrderPositionId = {id}`, including nested objects.

- Get order position by order ID
  `GET api/OrderPositions/Order/{id}`, where `{id}` is a valid Order ID
  Responds with the `OrderPositions` where the foreign key `OrderId = {id}`, including nested objects. This is used to get all the order positions of a specific order.

- Create order postion
  `POST api/OrderPositions`, with an `OrderPosition` object as request body
  Responds with status code `201` and an identical `OrderPosition` object with the `OrderPositionId` field set.

- Remove order position
  `DELETE api/OrderPositions/{id}`, where `{id}` is the ID of the `OrderPosition` to remove.
  Responds with status code `200` if the order position was removed. Response body contains a copy of the removed order position.

ProductController handles requests pertaining to products. It supports `GET` requests. These have not been fully tested, and for this reason, they are not included in this section.

To see more detailed documentation of the Order Service API, see Appendix D.

6.2.5 Shared Libraries

To increase code reusability, we packaged a number of key functions shared between services into shared libraries. This allows us to easily apply bug fixes across all of OXMES, and it makes it easy to start work on new services.

6.2.5.1 ServiceTypes

This library contains a single enum, `ServiceTypes`. This enum contains all currently known services and their type ID. These are:

1 Topology Service
2 Work Plan Service
3 Order Service
4 Operation Scheduling Service
5 Order Scheduling Service
6 Watchdog Service
7 User Management Service
8 Topology Discovery Service
9 Analytics Service
100 User Gateway
150 PLC Gateway
200 User Client

Please note, that not all of these have been implemented; some types are only defined now to ease future work.

6.2.5.2 ServiceUtilities

This library contains a number of utilities useful for services. Currently, it contains the C# representation of the `Service` model. Along with this is also a `ServiceClient` which currently wraps a standard implementation for registering with the User Gateway as a service.

6.2.5.3 EventBusClient

This library contains the tools necessary to use the Event Bus as a client. This includes reference implementations for common events, message type enums, a message validator to discard incorrectly formatted events, and a client that wraps all the necessary Event Bus functions, such as connecting to the Event Bus, sending events, receiving events and disconnecting from the Event Bus.

6.3 User Client

To control OXMES, a client with a graphical user interface (GUI) has also been developed. Unfortunately, as .NET Core does not yet support GUIs, this client has been implemented for .NET Framework (Windows-only) using Windows Presentation Foundation (WPF). We deem a Windows-only client acceptable because 1) it can easily be replaced, and 2) the current Festo MES client we are trying to replace is also Windows-only.

It is implemented as a single window, and uses tabs to neatly separate information. Thus, there is a tab for orders, showing only past, present and future orders,

along with their properties; and a tab for the topology, allowing the user to modify topologies, modules, resources and applications. At runtime, this client allows reconfiguring any detail of the topology.

Figures A.1, A.2, A.3, A.4 and A.5 show some screenshots of the client.

6.4 Event Bus

The event bus is specified as a bus all services are connected to, and where any service can, at any time broadcast an event, and every other service *should* be able to receive it. We have implemented this by having all services connect to the User Gateway with a persistent WebSocket. This allows bi-directional communication between the User Gateway and a service at all time during their respective lifetime. To broadcast an event, a service will transmit it to the User Gateway, and the User Gateway will then forward the event to all other interested services. To receive an event, a service must first subscribe to the event type with the User Gateway.
User Gateway acts as both the Event Bus server (see Section 6.2.1.2) and as a client on the Event Bus.

6.4.1 Protocol

Unlike the REST interface the User Gateway and the other services expose, which is based on HTTP, the WebSocket interface used for the Event Bus does not contain a well structured protocol. For this reason, we have created our own protocol on top of what WebSocket does contain to support sending events over WebSocket.

6.4.1.1 Connecting

When an Event Bus client connects, it has to complete three steps:

1. If the service hasn't already been announced, it must be announced to the User Gateway as a service.
2. Start a standard WebSocket connection to the Event Bus server (`ws://localhost:5050` by default)
3. Subscribe to messages of type *Service Shutdown* (see Section 6.4.1.3).

6.4.1.2 Events

Events transmitted over the Event Bus must be serialised as JSON. Events are to always have **all** the following fields:

Name	Type
messageUuid	string
originServiceUuid	string
severity	int
messageType	int
time	string
payload	any

If any of these fields are missing or wrongly formatted, services should discard the event, and the Event Bus Server will definitely discard the event.

messageUuid must be a string representing a UUID, generated by the client sending the event.

originServiceUuid must be a string representing the UUID of the service that sent the event.

severity must be an integer representing the severity of the event. This is used to allow more fine-grained control of event subscriptions. Known values are:

10	Trace
20	Debug
30	Info
40	Warn
50	Error
60	Fatal
1000	Control

messageType must be an integer representing the type of message. This is used for event subscriptions, and for determining how to deserialise the payload. Known values can be seen in Section 6.4.1.3.

time must be a string containing the ISO 8601 [13] formatted time at which the event was sent.

payload should be an object. The content of the object depends on the message type. A list of known events can be seen in Section 6.4.1.4.

6.4.1.3 Common Event Types

The Event Bus is programmed to be forward compatible. This is achieved by having a standard polymorphic format. The key in this format is the `messageType` property. This property denotes how data should be interpreted. Though it is optional which event types each service supports, there are some all must be able to support.

Type	Name
0	Undefined
10	Subscribe
1000	ServiceShutdown

It is important to note, that `0: Undefined` events should be considered errors and should be discarded.

6.4.1.4 Common Events

Though the Event Bus is programmed to be forwards compatible, supporting new types of events without any change to the Event Bus Server, there are some events that are standard, and that all services should understand.

Subscribe event is used by services to subscribe to different types of events. These events are not broadcasted by the Event Bus server.
The `severity` field in subscribe events must be `1000: Control`, and the `messageType` field must be `10`.
The format of `payload` must be an object with the following fields:

Name	Type
minimumSeverity	int
maximumSeverity	int
messageType	int

`messageType` represents the event type being subscribed to. `minimumSeverity` must be an integer representing the minimum severity of events to receive of type `messageType`, and `maximumSeverity` must be an integer representing the maximum severity. Both are included.
Listing 6.12 shows an example of a `Subscribe` event, subscribing to `ServiceShutdown` events.

```
{
    "messageUuid": "cbe1b41b-e91a-4fb7-a5b0-00b0fcdcc871",
    "originServiceUuid":
       "15074ecf-a86c-43ca-bfe2-b98a81c8e4b4",
    "severity": 1000,
    "messageType": 10,
    "time": "2018-06-26T08:00:00Z",
    "payload": {
        "minimumSeverity" = 30,
        "maximumSeverity" = 1000,
        "messageType": 1000
    }
}
```

Listing 6.12: An example of a `Subscribe` event message

ServiceShutdown event is used by a service to announce to other services on the Event Bus, that it is shutting down, to allow other services to clear any cached information about this service they might have.
The `payload` of these types of events must be a string representation of the UUID of the service that will shut down. If a service disappears unannounced the User Gateway may broadcast a `ServiceShutdown` event on its behalf, but for all other services, `originServiceUuid` and `payload` must be the same.
Listing 6.13 shows an example of `ServiceShutdown`.

```
{
    "messageUuid": "cbe1b41b-e91a-4fb7-a5b0-00b0fcdcc871",
    "originServiceUuid":
       "15074ecf-a86c-43ca-bfe2-b98a81c8e4b4",
    "severity": 1000,
    "messageType": 10,
    "time": "2018-06-26T08:00:00Z",
    "payload": "15074ecf-a86c-43ca-bfe2-b98a81c8e4b4"
}
```

Listing 6.13: An example of a `ServiceShutdown` event message

6.4.1.5 Disconnecting

The Event Bus disconnect procedure is split into two parts. First, a disconnecting Event Bus client must announce on the Event Bus that it is disconnecting. This

is done by broadcasting a `ServiceShutdown` event. This will let other services know that the service is going to disappear. Then the service must do a two-way close handshake, as defined by the WebSocket specification. This means that before a service can completely close its Event Bus connection, it must wait for the Event Bus Service to acknowledge that the connection will be closed.

6.5 Evaluation of the technical choices

In this section we review some of the technical choices we made, and see how well they fit the purpose of the project.

6.5.1 C# language

Using C# as the development language for this project was a good idea. The fact that one of the members of the group had previous experience with it helped a lot. There were hardly any obstacles with the syntax, except the syntactic sugar used in C# which can occasionally differ from other languages. One example of this is the null-coalescing (`??`) operator in C# which does not have an equivalent in other languages. Listing 6.14 shows an example of a null-coalescing operator in action.

```
1 // Set y to the value of x if x is NOT null; otherwise,
2 // if x == null, set y to -1.
3 int y = x ?? -1;
```

Listing 6.14: Example of a null-coalescing operator in use [7]

Another aspect worth mentioning, that required some investigation, was the use of lambda expressions (an example can be seen in Listing 6.15), which are not very popular in the embedded world.

```
40 var order =  _context.Orders
41     .Include(op => op.OrderPositions)
42     .Where(i => i.OrderId == id);
```

Listing 6.15: Lambda expression in `OrdersController.cs`. Here, `Include` and `Where` are applied on each instance of `OrderPositions`, respectively `OrderId`

6.5.2 ASP.NET Core

Overall, the choice of ASP.NET Core as the framework to build the services around has proven a good choice. The simple way to create code to respond to a request in a RESTful manner, jump started the development of OXMES. It has also allowed

us to work on the micro service architecture, without having a deep knowledge about it beforehand.
But while the documentation has generally been very helpful, with nice tutorials maintained by Microsoft, implementing user management turned out to be too difficult to accomplish in our time frame. This was largely due to the fact that the existing documentation and tutorials were all based on implementing user management for a web app for its own Web UI, and for a single-service architecture, and the examples were not transferable to our architecture.

In ASP.NET Core, there can sometimes be many ways to accomplish the same task, one example of this is with *Fluent API* versus *Data Annotations* in EF Core. This proved both a strength and a weakness for us. With multiple ways to accomplish the same task, it was often possible to find a way that fit *us*. But at the same time, when searching for a solution to a problem, it could make it difficult to find something that fit out problem exactly.

6.5.3 Entity Framework Core with *code first*

The choice for generating the databases for the services using EF Core was a good one, but it did not come without cost: some of the quirks of EF Core has cost us significant time to discover and work around.
Using EF Core means that the model only has to be written once, either as code or as a database. Using *code first* meant that we never had to leave the comfort of C# to enter SQL. This sped up development significantly. Additionally, the fact that any changes to the model can quickly be reflected in the database, even with automatic migration of data, meant that we could add properties to our model with ease.

Unfortunately, it turned out that our model in many situations needed to adapt to EF Core. So in effect, the shape of the model, is largely dictated by what EF Core expects. A clear example of this are all the relationships. Both parties in a one-to-one relationship should ideally have references to each other, something that causes problems with the default JSON serialisation. And while EF Core can recognise and model one-to-one and one-to-many relationships by itself, many-to-many relationships require extra classes to model the associative tables needed in the database. Ideally, database logic should be hidden from clients, so these associative table classes should be hidden. This means that classes having many-to-many relationships with other classes may need to implement special serialisation logic.

Another development speed bump is the way references must be loaded at runtime. By default, when searching EF Core for an entity, it will not include data from references; when searching for entities in EF Core, you have to manually specify in

the query which relationships should be included in the response.
In Listing 6.15 you can see, how it has been manually specified that information from the `OrderPositions` property should be included with the result.

6.5.4 Database server

Considering the fact that OXMES will have several services that need to store data, a question that arises is:

Is it better to have a single database server with several databases, or multiple instances of an database server?

The main argument for running multiple database servers is that services will be very strictly isolated from each other; the failure of one does not directly influence others. Likewise, if a database server itself fails, only a single service will fail too. However, until now, this has not been a concern for the project, as the 2 implemented services can easily share a single database server, and since both services are critical for operation, so if one fails, the other might as well fail too. In the future, it should be considered that OXMES will run on a less powerful PC, which might not provide the performance necessary to host multiple database servers. Additionally, it should be considered that reducing the number of database servers will also decrease the amount of work necessary to maintain them, especially when you reduce the number of different *types* of database servers.
Also, as it is the case that some of OXMES's services might need to use each others data, then the second scenario might result in unnecessary overhead.

6.5.5 API documentation

For the purpose of writing the API documentation, we used the Swashbuckle [5] implementation of Swagger [12] for ASP.NET Core. Out of the bundle of tools Swagger provides, the ones we used the most were Swagger UI and the automatic documentation generation. The use of Swagger was trivial, once the proper toolchain is set up. The documentation itself is pulled from the existing C# XML Documentation as seen in Listing 6.16.

```
37 /// <summary>
38 /// Gets individual orders
39 /// </summary>
40 /// <param name="id">The ID of the order</param>
41 /// <returns>A json object containing one order and its
42 /// components (order positions, respectively
   products)</returns>
```

Listing 6.16: Comments describing a GET call in `OrdersController.cs`

Swagger will then take these comments and create a JSON file, describing the entire API. This is mainly used for Swagger's Web UI, but while the server is running, we can get a `swagger.json` file and use it to generate a different file in the format we want. As seen in Listing 6.17, besides using the comments we provided, Swagger also uses information it can find in the code, i.e. the *parameters* of a function, and the *responses* it produces.

```
"/api/Orders/{id}": {
    "get": {
        "tags": [
            "Orders"
        ],
        "summary": "Gets individual orders",
        "operationId": "ApiOrdersByIdGet",
        "consumes": [],
        "produces": [],
        "parameters": [
            {
                "name": "id",
                "in": "path",
                "description": "The ID of the order",
                "required": true,
                "type": "integer",
                "format": "int32"
            }
        ],
        "responses": {
            "200": {
                "description": "Success"
            }
        }
    },
    ⋮
}
```

Listing 6.17: Snippet of the JSON file, containing the same function description as in Listing 6.16

The Markdown file is generated from the `swagger.json` file, using the CLI tool called `swagger2markup` [4]. The reason for wanting Markdown as the final product (Figure 6.6), is that it is portable and flexible. We can, for example, use the Markdown file to generate a PDF. But most importantly, all web-based repository managers provide tools for displaying Markdown, thus making it easy for other developers to use and read the API documentation.

Gets individual orders

`GET /api/Orders/{id}`

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the order	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content

Tags

- Orders

Figure 6.6: Snippet of the Markdown view, containing the same function as in Listing 6.16

A concrete example of an XML Documentation to Swagger to Markdown toolchain, is the following, which can export the API documentation of Topology Service:

```
@echo off
mkdir TopologyService
cd TopologyService
del swagger.json
ubuntu run wget http://localhost:5051/swagger/v1/swagger.json
cd ..
java -jar swagger2markup-cli-1.3.3.jar convert -c
   swagger2markdown.properties -i
   TopologyService/swagger.json -f
   TopologyService/TopologyService
```

Listing 6.18: A batch script to pull swagger.json from Topology Service and convert it to markdown

CHAPTER 7

Evaluation

This chapter will cover the evaluation of OXMES, in the state it was at the time of handing in this report. The evaluation was done by talking with our contacts from the Department of Materials and Production (both a user of the MES - Casper Schou; and one of the computer science researchers working there - Chen Li).

7.1 Usability of OXMES

After looking at the User Interface of OXMES Casper from MP provided feedback on it. The following are some of the aspects he mentioned (paraphrased):

> *The views of the services look good, and it is nice that you have all the configuration settings for the topology in one place.*

Among the suggestions for the view, there were:

> *It would be nice to have a view displaying the overall systems status. This could contain the connected modules, their latest status, or statistics of processing times for their applications.*
> *We should continue to keep the topology related settings in one place, where they are easy to overview.*
> *A cool thing that could be done is to have a picture of the application existing on top of a conveyor, together with a short description. This would make the Application tab more intuitive to use.*

If implemented, the last suggestion could look as shown in Figure 7.1.

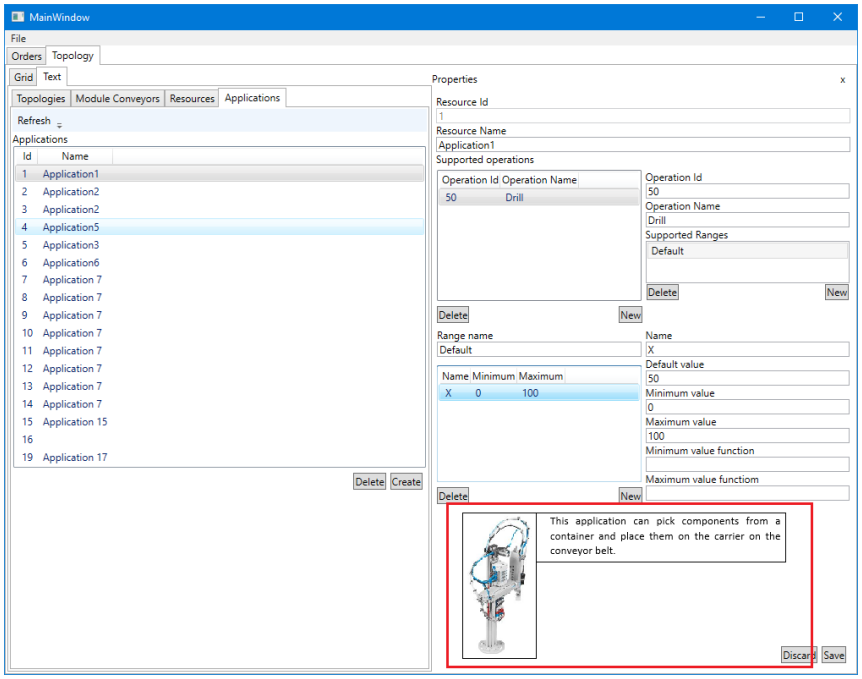


Figure 7.1: Applications tab. In the red square can be seen a picture and a short description of an application. The placement is not final.

7.2 Functionality and features of OXMES

In order to get feedback on the functionality of OXMES, the work done was presented to Casper, and then he mentioned the aspects he liked, and the ones that need improvement. The following is what he had to say about it (paraphrased):

> *It is nice that multiple topologies can be stored in a database, and that inactive topologies can be achieved.*
> *In the Application tab, it is really nice that we can define the operations. This comes to prove that the separation between a operation and Application is important.*
> *The application ranges look good.*

The following are suggestions for OXMES:

> *It would be nice to be able to see average/actual processing times of an application.*
> *It would be nice to annotate the modules with their type. This is due to the fact that the current MES does not know what the modules look like and what they really do.*
> *It could be interesting to have a list of Known Errors, and keep track of how*

often they occur. Based on these, further documentation could be written on how to address them.

7.3 API documentation

The feedback received from Chen acknowledged the API documentation as generally good. Due to a communication mistake, he did not receive the documentation for the User Gateway, thus we did not get any specific feedback on this service. Among the things he proposed are (paraphrased):

> *It would be a good idea to add the version number of an API in the URL. For example,* `/api/V1/Applications`*.*
> *Make sure not to violate consistency when you update an element. This has to do with data structures containing foreign keys of other entities.*
> *Some of the expected properties of a model are missing (i.e.* `plannedEndTime` *of an* `Order`*). Make sure you provide an explanation for this.*

As suggested, we take care to preserve consistency when updating structures containing foreign keys.
In Section 6.2.4.1 we present an explanation for why `plannedEndTime` is missing.

7.4 Extensibility

Throughout the development of OXMES, extensibility has been a key aspect. This has driven the implementation to focus on forward compatibility. This is evident in the User Gateway with the service discovery feature, and in the Event Bus. For both of these, `ServiceType` and `MessageType` respectively are implemented as integers, not enums. For services it means that new services with previously unknown types can connect, without necessitating changes to the User Gateway. And as for the Event Bus, because User Gateway more often than not just acts as a proxy, it does not need to interpret the event, and thus it can support new types of events, provided these new types of events follow the format defined in Section 6.4.1.2. These are just examples of how extensibility has been built into the code.
Another way OXMES has been made more extensible, is by packing common functionality into shared libraries. This gives a head start to any new services build in a Common Language Runtime (CLR) language (C#, F#, Visual Basic, etc.).

Despite this, there are still some limitations to easy extensibility. While services can easily be created and added to the architecture, if the user clients or the PLCs need to communicate with it, the User- and/or the PLC Gateway must be updated with new controllers to support the new services.

7.5 Known issues

One known issue with the current model is that, we model module conveyors. These are actually part of a pair of module conveyors packed into a single module. However, the two module conveyors in a module do not share any programmable logic controller (PLC), so from the perspective of the software, there is no way to know which two module conveyors are paired in the same module. For this reason, the entire topology of the Festo MMS is represented as a set of circular dependencies, regardless of their physical location and relationships. This makes it impossible to draw a graphical replica of the MMS in software.

Another known issue with the current model is our assumption that all module conveyors have only a single input port is wrong. While regular modules consists of two module conveyors each with just one input port and one output port, and each controlled with their own PLC, the branching modules uses a single PLC to control both conveyors. This means that branching modules consists of two input ports. This makes these branching modules impossible to represent in the current model.

Another minor issue that we found out during the evaluation, is that what we have called module conveyors are actually referred to as stations by MP.

CHAPTER 8

Conclusion

The overall goal of this master thesis was to look into improving approachability of the Modular Manufacturing Systems (MMS), at the Department of Materials and Production (MP) of Aalborg University. By first understanding how it works, several areas of interest were discovered, in which topics from the computer science field could be applied. The Manufacturing Execution System (MES) is one of the elements that is vital for operating an MMS. It needs to be highly customizable and to provide a way for the user to control and interact with the system. That being said, the MES offered an interesting challenge, where we could use oue knowledge and further expand it by exploring its problems, and implementing solutions for these.

This report is the written result of the activities carried out throughout the 10^{th} semester of the Embedded Software Systems Master programme. During this time, the work started in the 9^{th} semester was continued, while tackling a different set of problems, related to implementing OXMES. In the Problem Statement (Chapter 4) several research questions have been formulated, with the intention to have them answered in the different sections of this report. This was done by investigating these topics, coming up with observations and consulting our contacts both at the Department of Materials and Production and the Department of Computer Science. After designing and implementing a solution, we facilitated communication to receive feedback, mainly from the future users of OXMES. This way, we were able to evaluate both our decisions and the usability of the final product.

The following are the research questions, as they have been addressed throughout the report:

How can we facilitate that the system being built will live on?
The duty of OXMES is slightly more complex than that of a MES in a traditional setup (i.e. in a factory). This is because OXMES will need to evolve in sync with the activities at MP. Compared to a traditional MES, OXMES should be able to:

- Facilitate experiments. This means highly customizable features
- Accommodate new services
- Interchange existing services with different implementations of the same service
- Provide a simple overview of the system

For this reason, we tried to implement OXMES using tools that offer a high degree of flexibility. This way, we hope that the system will be easy to adapt to the needs of MP. Furthermore, based on this report, the inner workings of OXMES should be easy to understand. API documentation for the services implemented are included in the appendices of this report. By looking at all these things, we consider that is should be relatively trivial to continue developing OXMES.

Have we followed the initial system design and user requirements?
We have followed the system design and user requirements set in the 9^{th} semester. From the system design we have partially implemented:

- A user interface
- The User Gateway
- The event bus
- The Topology Service
- The Order Service

The requirements list has evolved very little in the 10^{th} semester. For this reason there were no misunderstanding as to what the system should be able to do. From the initial requirement list, features worth mentioning of OXMES are:

- Modular software architecture
- Resource topology description
- Database of past, current and future orders
- Order API
- Orders of multiple types of products
- Runnable on desktop workstation

Do we comply with the user requirements?
As shown in the Evaluation (Chapter 7) we comply with the user requirements relevant to the features currently implemented in OXMES.

How will the end result be evaluated?
The end result was evaluated in terms of usability, functionality and extensibility in Evaluation (Chapter 7). This was mainly done by talking with the future users of OXMES. Based on their feedback, we could reflect on the choices taken during the project, and understand the impact they have on the end user.

What is the conclusion of this project and what remarks are there for future work?
This project concluded in having partially implemented the initial system design of OXMES. Taking into consideration the time frame of the project, we expected not to be able to have a fully fledged MES by the end of the semester, and for this reason, we prioritised the implementation of OXMES's modules accordingly. Based on this, we believe that we have reached our initial goals.
Remarks, ideas and suggested new features are discussed in the following section.

8.1 Perspectives

The work on OXMES is not yet completely done, and for this reason, the next step would be the implementation of the remaining services and features.

A limitation to the extensibility of OXMES is the occasional need to update the User- and/or PLC Gateway when new services are created. Ideally these gateways should be generalised further, to reduce this burden. An idea is to build in a plug-in framework for new services.

During the discussions with MP, a couple of new suggestions arose, that could be interesting to research in the future:

- The possibility of interfacing the OXMES with other types of equipment than the original Festo modules. An example of new modules could be mobile robots. The challenge here is of course the modelling of these robots.
- These mobile robots could be represented as an independent module with a drop off point connected to a module in an existing topology. But of course, other solutions should be explored.
- The ability to schedule an outside robot (mobile or not mobile) to pick up things, and integrate it with an existing topology. This suggestion comes as an extension to the previous one. It is impossible, at the time being to include non Festo modules in the current MES configuration.
- Have an expected finished-by time to allow scheduling for a product pick up. This can be achieved by being able to better keep track of the lifetime

of the orders which are in the progress of being produced, coupled with an understanding of processing times of pending operations.

Furthermore, it would be interesting to check the progress of the project that worked on supporting parallel scheduling production in the MMS [6]. Among the goals of this project, there was the idea that the Festo MMS should be able to support parallel production, if the system has several modules that can do the same operation. This is one thing that can be integrated and physically tested with the use of the Operation Scheduling Service of OXMES.

As seen in Section 6.2.2, we did not succeeded in implementing a User Management Service, to authenticate users. One of the reasons for this, is the lack of experience in system security. There are however, students who specialise in this topic, who could help with advice and suggestions. One such group of students is actually working on a related topic on the Festo MMS. Their project for the past semester dealt with making sure that unauthorised MMS modules (or devices emulating these) cannot be attached to the Festo MMS.

Bibliography

[1] S. Cavalieri et al. "A web-based platform for OPC UA integration in IIoT environment". In: *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (2017), pp. 1–6.

[2] OPC Foundation. *Official OPC UA .Net Standard Stack and Samples from the OPC Foundation*. https://github.com/OPCFoundation/UA-.NETStandard.

[3] OPC Foundation. *OPC Unified Architecture*. http://www.opcfoundation.org/.

[4] Github. *Robert Winkler*. https://github.com/Swagger2Markup/swagger2markup.

[5] Github. *Swashbuckle*. https://github.com/domaindrivendev/Swashbuckle.

[6] Martin Kristjansen. *Supporting Parallel Production in Festo*. Student project. Department of Computer Science, School of Information and Communication Technology, Aalborg University. 2017.

[7] Microsoft. *?? Operator (C# Reference)*. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-coalescing-operator.

[8] Microsoft. *ASP.NET MVC Overview*. https://docs.microsoft.com/en-us/previous-versions/aspnet/web-frameworks/dd381412(v=vs.108).

[9] Microsoft. *Creating a Model*. https://docs.microsoft.com/en-us/ef/core/modeling.

[10] Anders Normann Poulsen and Gabriel Vasluianu. *Extensible Manufacturing Execution System - Designing a modular, extensible MES for dependability and research*. Student project. Department of Computer Science, School of Information and Communication Technology, Aalborg University. 2018.

[11] Festo Media Service. *Festo embeds OPC-UA in its valve terminals to drive benefits of Industry 4.0*. https://www.festo.com/net/en-gb_gb/SupportPortal/Details/404800/PressArticle.aspx.

[12] SmartBear Software. *Swagger*. https://swagger.io/.

[13] Wikipedia. *ISO 8601*. https://en.wikipedia.org/wiki/ISO_8601.

APPENDIX A

User Client Screenshots

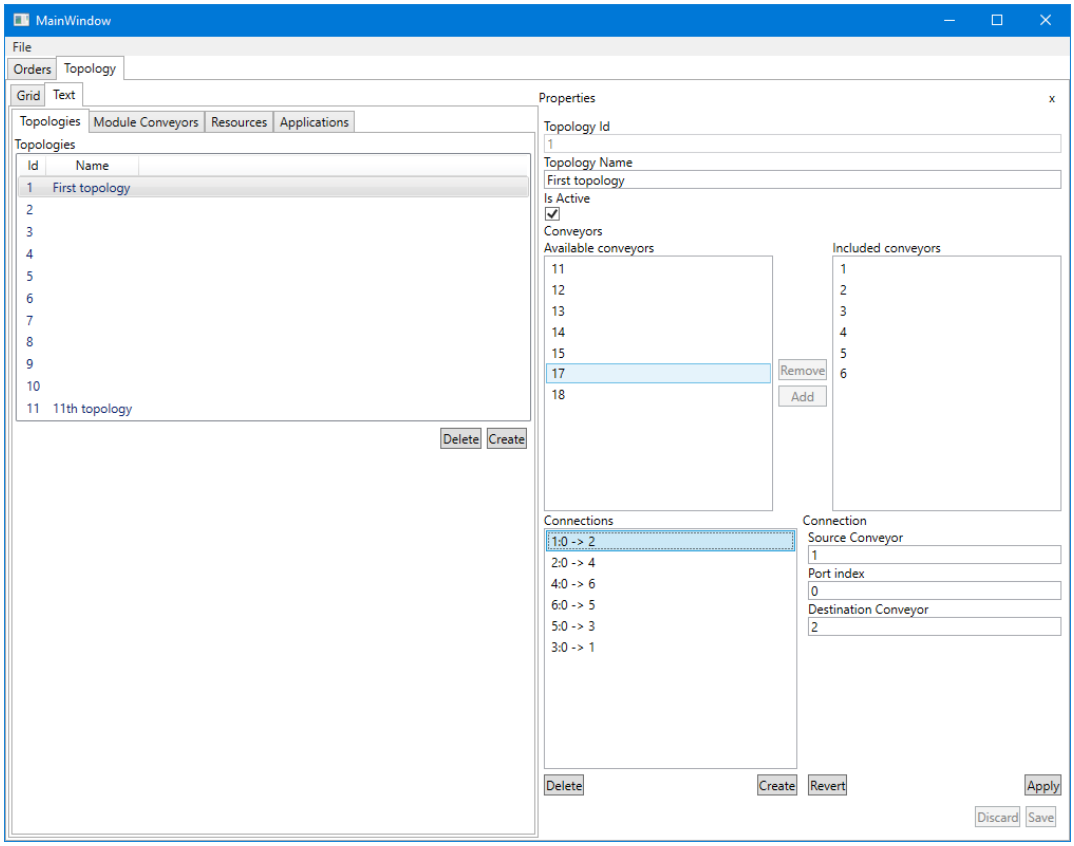


Figure A.1: Topologies tab

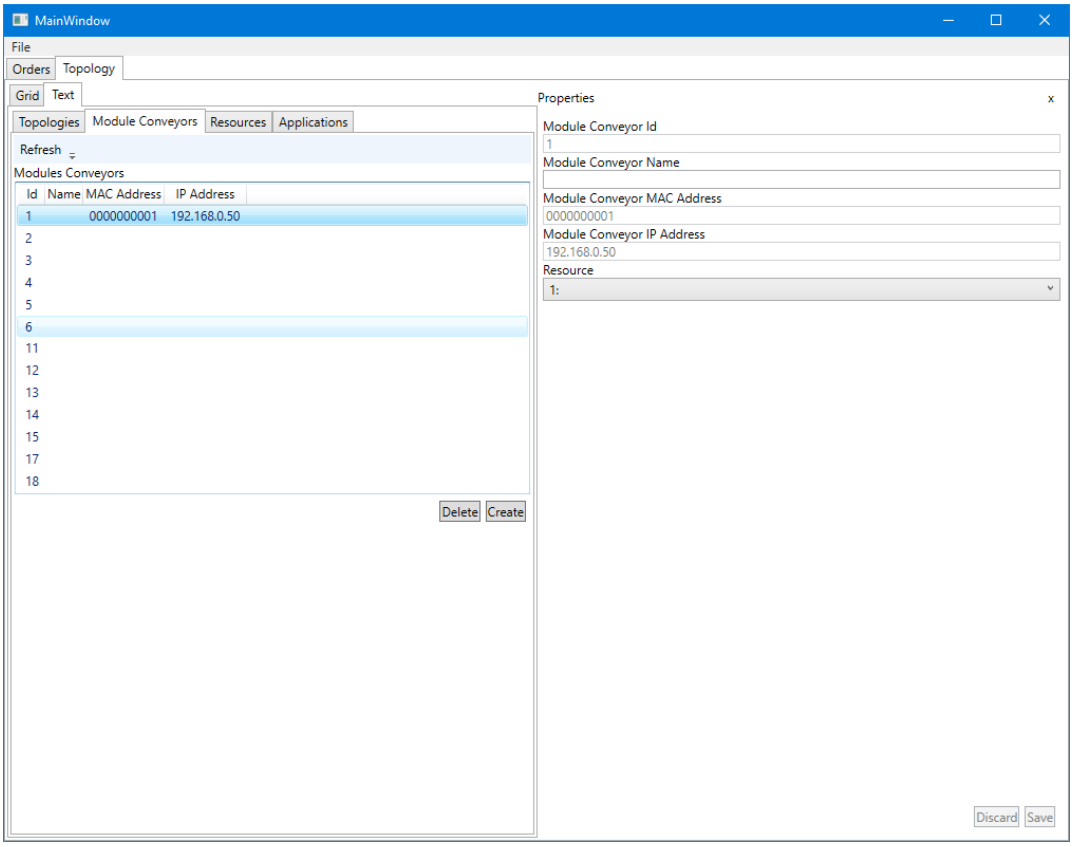


Figure A.2: Module Conveyors tab

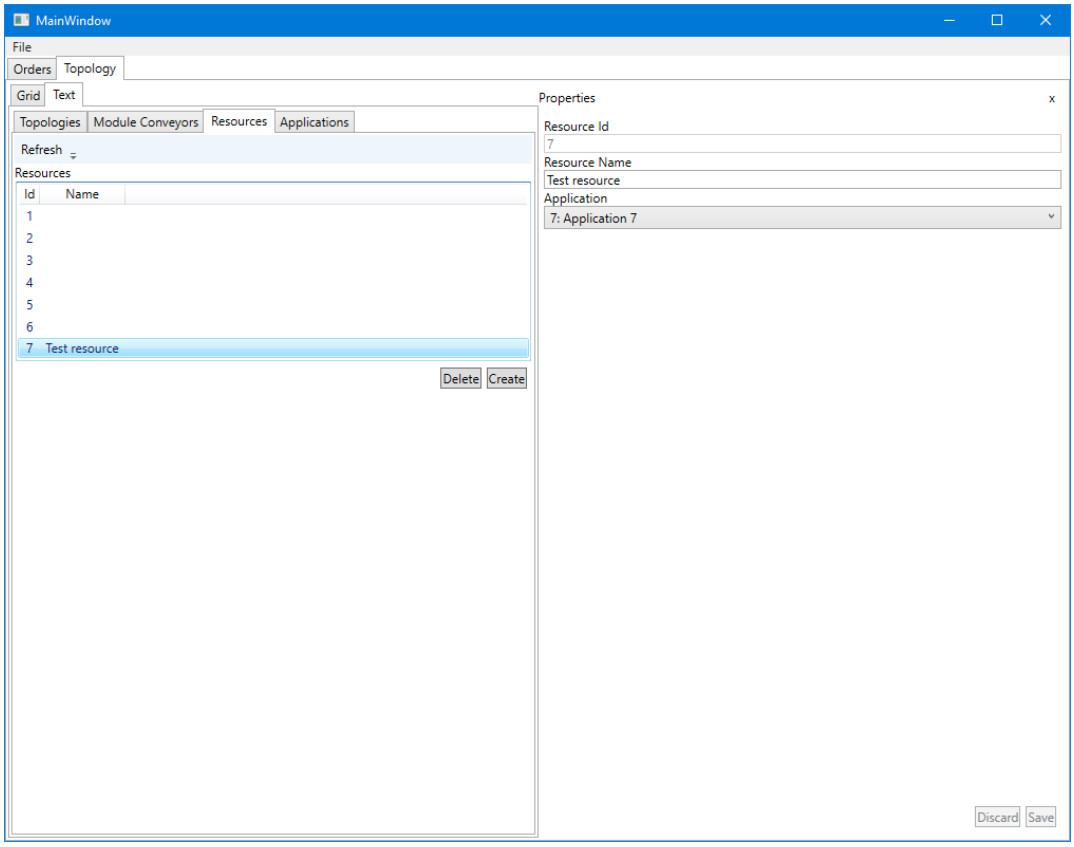


Figure A.3: Resources tab

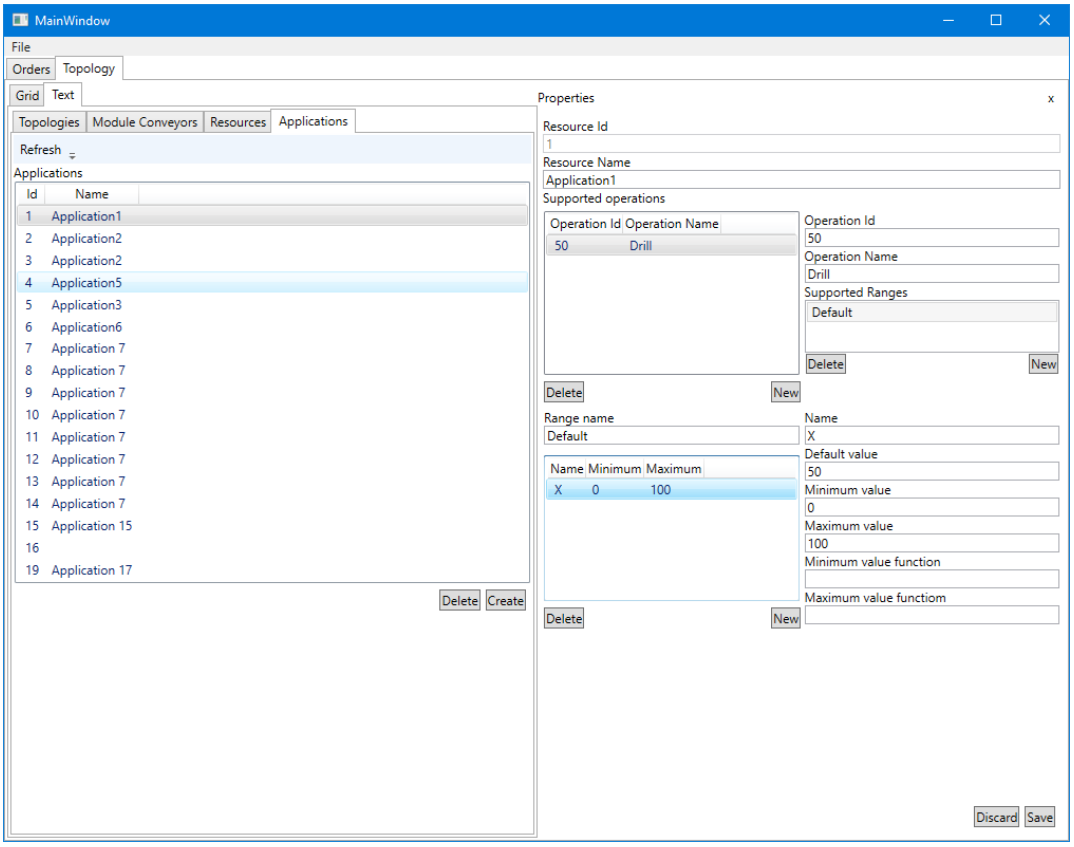


Figure A.4: Applications tab

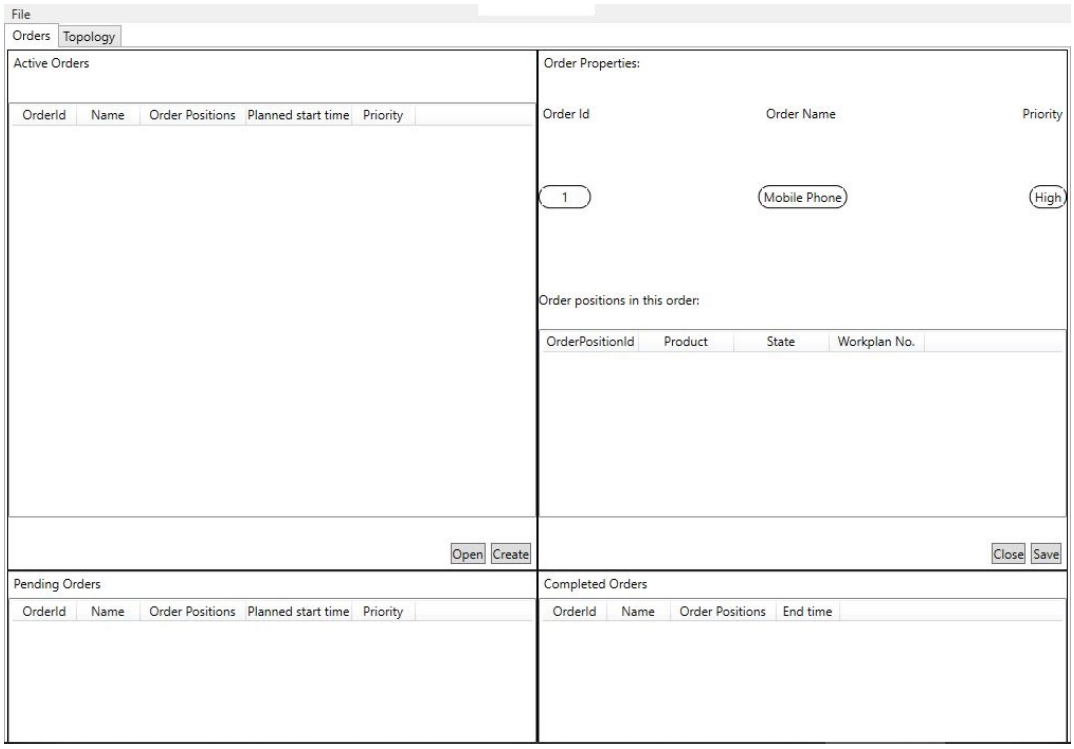


Figure A.5: Order view

APPENDIX B

User Gateway API Documentation

User Gateway API

Overview

Version information

Version : v1

Paths

Create a new Application

```
POST /api/Applications
```

Description

For more information about Application, see documentation of Topology Service

Example of application:

```
POST /api/Application
{
    "applicationId": 0,
    "applicationName": "name",
    "resourceId": 0,
    "supportedOperations": [
        {
            "applicationOperationId": 0,
            "operationId": 0,
            "operationName": "name",
            "supportedRanges": [
                {
                    "id": 0,
                    "name": "name",
                    "parameters": [
                        {
                            "id": 0,
                            "name": "name",
                            "minimum": 0,
                            "maximum": 0,
                            "minimumFunction": "{X}*2",
                            "maximumFunction": "{X}*4",
                            "defaultValue": 0
                        }
                    ]
                }
```

```
            ]
        }
    ]
}
```

Parameters

Type	Name	Description	Schema
Body	**application** *optional*	The Application	object

Responses

HTTP Code	Description	Schema
201	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Applications

Get a shallow list of Applications

```
GET /api/Applications
```

Description

For more information about Application, see documentation of Topology Service

Example output:

```
GET /api/Applications
[
    {
        "applicationId": 0,
        "applicationName": "name",
        "resourceId": 0,
        "supportedOperations": null
    }
]
```

Responses

HTTP Code	Description	Schema
200	Success	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Tags

- Applications

Get a deep copy of an Application

```
GET /api/Applications/{id}
```

Description

For more information about Application, see documentation of Topology Service

Example output:

```
GET /api/Application/{id}
{
    "applicationId": 0,
    "applicationName": "name",
    "resourceId": 0,
    "supportedOperations": [
        {
            "applicationOperationId": 0,
            "operationId": 0,
            "operationName": "name",
            "supportedRanges": [
                {
```

```
                    "id": 0,
                    "name": "name",
                    "parameters": [
                        {
                            "id": 0,
                            "name": "name",
                            "minimum": 0,
                            "maximum": 0,
                            "minimumFunction": "{X}*2",
                            "maximumFunction": "{X}*4",
                            "defaultValue": 0
                        }
                    ]
                }
            ]
        }
    ]
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the Application to get	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Tags

- Applications

Update an Application

```
PUT /api/Applications/{id}
```

Description

For more information about Application, see documentation of Topology Service

Example of application:

```
PUT /api/Application/{id}
{
    "applicationId": 0,
    "applicationName": "name",
    "resourceId": 0,
    "supportedOperations": [
        {
            "applicationOperationId": 0,
            "operationId": 0,
            "operationName": "name",
            "supportedRanges": [
                {
                    "id": 0,
                    "name": "name",
                    "parameters": [
                        {
                            "id": 0,
                            "name": "name",
                            "minimum": 0,
                            "maximum": 0,
                            "minimumFunction": "{X}*2",
                            "maximumFunction": "{X}*4",
                            "defaultValue": 0
                        }
                    ]
                }
            ]
        }
    ]
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	A valid Application ID	integer (int32)
Body	**application** *optional*	The updated Application	object

Responses

HTTP Code	Description	Schema
204	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Applications

Delete an Application

```
DELETE /api/Applications/{id}
```

Description

For more information about Application, see documentation of Topology Service

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the Application to delete	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content

HTTP Code	Description	Schema
503	No Topology Service	No Content

Tags

- Applications

Create a module conveyor

```
POST /api/ModuleConveyors
```

Description

For more information about the ModuleConveyor model, see the Topology Service documentation

Example of module conveyor:

```
POST /api/ModuleConveyors
{
    "moduleConveyorId": 0,
    "moduleConveyorName": "unique-name",
    "plcIpAddressString": "10.0.0.144",
    "plcMacAddressString": "0011223344",
    "resourceId": 0
}
```

Parameters

Type	Name	Description	Schema
Body	**moduleConveyor** *optional*	The new module conveyor	object

Responses

HTTP Code	Description	Schema
201	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Consumes

- application/json-patch+json
- application/json
- text/json
- application/*+json

Tags

- ModuleConveyors

Get all module conveyors

```
GET /api/ModuleConveyors
```

Description

For more information about the ModuleConveyor model, see the Topology Service documentation

Example output:

```
GET /api/ModuleConveyors
[
    {
        "moduleConveyorId": 0,
        "moduleConveyorName": "unique-name",
        "plcIpAddressString": "10.0.0.144",
        "plcMacAddressString": "0011223344",
        "resourceId": 0
    },
    {
        "moduleConveyorId": 1,
        "moduleConveyorName": "special-name",
        "plcIpAddressString": "10.0.0.145",
        "plcMacAddressString": "0011223355",
        "resourceId": 1
    }
]
```

Responses

HTTP Code	Description	Schema
200	Success	No Content
502	Server Error	No Content

HTTP Code	Description	Schema
503	No Topology Service	No Content

Tags

- ModuleConveyors

Get a module conveyor by ID

```
GET /api/ModuleConveyors/{id}
```

Description

For more information about the ModuleConveyor model, see the Topology Service documentation

Example output:

```
GET /api/ModuleConveyors/{id}
{
    "moduleConveyorId": 0,
    "moduleConveyorName": "unique-name",
    "plcIpAddressString": "10.0.0.144",
    "plcMacAddressString": "0011223344",
    "resourceId": 0
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the module conveyor	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Tags

- ModuleConveyors

Update a ModuleConveyor

```
PUT /api/ModuleConveyors/{id}
```

Description

For more information about the ModuleConveyor model, see the Topology Service documentation

Example of module conveyor:

```
POST /api/ModuleConveyors/{id}
{
    "moduleConveyorId": 0,
    "moduleConveyorName": "unique-name",
    "plcIpAddressString": "10.0.0.144",
    "plcMacAddressString": "0011223344",
    "resourceId": 0
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	A valid ID of the ModuleConveyor	integer (int32)
Body	**moduleConveyor** *optional*	The ModuleConveyor	object

Responses

HTTP Code	Description	Schema
204	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- ModuleConveyors

Delete a ModuleConveyor

```
DELETE /api/ModuleConveyors/{id}
```

Description

For more information about the ModuleConveyor model, see the Topology Service documentation

Example output:

```
DELETE /api/ModuleConveyors/{id}
{
    "moduleConveyorId": 0,
    "moduleConveyorName": "unique-name",
    "plcIpAddressString": "10.0.0.144",
    "plcMacAddressString": "0011223344",
    "resourceId": 0
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	A valid ID of the ModuleConveyor to delete	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content

HTTP Code	Description	Schema
503	No Topology Service	No Content

Tags

- ModuleConveyors

Ping the service

```
GET /api/Ping/{pingMessage}
```

Description

This call can be used by other services to validate that User Gateway is still alive and running

Parameters

Type	Name	Description	Schema
Path	**pingMessage** *required*	An arbitraty string to return	string

Responses

HTTP Code	Description	Schema
200	Success	string

Produces

- `text/plain`
- `application/json`
- `text/json`

Tags

- Ping

Create a Resource

```
POST /api/Resources
```

Description

For more information about the Resource model, see the Topology Service documentation

Example output:

```
POST /api/Resources
{
    "applicationId": 0,
    "moduleConveyorId": 0,
    "resourceId": 0,
    "resourceName": "Name"
}
```

Parameters

Type	Name	Description	Schema
Body	**resource** *optional*	The Resource to create	object

Responses

HTTP Code	Description	Schema
201	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Resources

Get a list of all Resources

```
GET /api/Resources
```

Description

For more information about the Resource model, see the Topology Service documentation

Example output:

```
GET /api/Resources
[
    {
        "applicationId": 0,
        "moduleConveyorId": 0,
        "resourceId": 0,
        "resourceName": "Name"
    },
    {
        "applicationId": 1,
        "moduleConveyorId": 2,
        "resourceId": 1,
        "resourceName": "Name"
    }
]
```

Responses

HTTP Code	Description	Schema
200	Success	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Tags

- Resources

Get a Resource by ID

```
GET /api/Resources/{id}
```

Description

For more information about the Resource model, see the Topology Service documentation

Example output:

```
GET /api/Resources/{id}
{
    "applicationId": 0,
    "moduleConveyorId": 0,
    "resourceId": 0,
    "resourceName": "Name"
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The Resource ID	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Tags

- Resources

Update a Resource

```
PUT /api/Resources/{id}
```

Description

For more information about the Resource model, see the Topology Service documentation

Example output:

```
PUT /api/Resources/{id}
{
    "applicationId": 0,
    "moduleConveyorId": 0,
```

```
    "resourceId": 0,
    "resourceName": "Name"
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the Resource to update	integer (int32)
Body	**resource** *optional*	The updated Resource	object

Responses

HTTP Code	Description	Schema
204	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Resources

Delete a Resource

```
DELETE /api/Resources/{id}
```

Description

For more information about the Resource model, see the Topology Service documentation

Example output:

```
DELETE /api/Resources/{id}
{
    "applicationId": 0,
    "moduleConveyorId": 0,
    "resourceId": 0,
    "resourceName": "Name"
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the Resource to delete	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Tags

- Resources

Announce a Service

```
POST /api/Service
```

Description

Example output:

```
POST /api/Services
{
    "uuid": "123e4567-e89b-12d3-a456-426655440000",
    "name": "name",
    "serviceType": 0,
```

```
    "isActive": true,
    "displayName": "Name",
    "serviceAddress": "http://localhost:5050/"
}
```

Parameters

Type	Name	Description	Schema
Body	**service** *optional*	The Service to announce	Service

Responses

HTTP Code	Description	Schema
201	Success	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Service

Get a list of all Services

```
GET /api/Service
```

Description

Example output:

```
GET /api/Services
[
    {
        "uuid": "123e4567-e89b-12d3-a456-426655440000",
        "name": "name",
        "serviceType": 0,
        "isActive": true,
        "displayName": "Name",
        "serviceAddress": "http://localhost:5050/"
```

```
    },
    {
        "uuid": "123e4567-e89b-12d3-a456-426655440001",
        "name": "name1",
        "serviceType": 1,
        "isActive": true,
        "displayName": "Name 1",
        "serviceAddress": "http://localhost:5051/"
    }
]
```

Responses

HTTP Code	Description	Schema
200	Success	No Content

Tags

- Service

Get Service by name

```
GET /api/Service/name/{serviceName}
```

Description

Example output:

```
GET /api/Services/name/{serviceName}
{
    "uuid": "123e4567-e89b-12d3-a456-426655440000",
    "name": "serviceName",
    "serviceType": 0,
    "isActive": true,
    "displayName": "Name",
    "serviceAddress": "http://localhost:5050/"
}
```

Parameters

Type	Name	Description	Schema
Path	**serviceName** *required*	The name of the requested Service	string

Responses

HTTP Code	Description	Schema
200	Success	No Content
404	Not Found	No Content

Tags

- Service

Get the active Service by service type

```
GET /api/Service/type/{typeId}
```

Description

Example output:

```
GET /api/Services/type/{typeId}
{
    "uuid": "123e4567-e89b-12d3-a456-426655440000",
    "name": "name",
    "serviceType": 0,
    "isActive": true,
    "displayName": "Name",
    "serviceAddress": "http://localhost:5050/"
}
```

Parameters

Type	Name	Description	Schema
Path	**typeId** *required*	The Service type	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
204	No active service of type typeId	No Content

Tags

- Service

Get a Service by UUID

```
GET /api/Service/{uuid}
```

Description

Example output:

```
GET /api/Services/{uuid}
{
    "uuid": "123e4567-e89b-12d3-a456-426655440000",
    "name": "name",
    "serviceType": 0,
    "isActive": true,
    "displayName": "Name",
    "serviceAddress": "http://localhost:5050/"
}
```

Parameters

Type	Name	Description	Schema
Path	**uuid** *required*	The UUID of the requested Service	string (uuid)

Responses

HTTP Code	Description	Schema
200	Success	No Content
404	Not Found	No Content

Tags

- Service

Updated a Service

```
PUT /api/Service/{uuid}
```

Description

Example output:

```
PUT /api/Services/{uuid}
{
    "uuid": "123e4567-e89b-12d3-a456-426655440000",
    "name": "name",
    "serviceType": 0,
    "isActive": true,
    "displayName": "Name",
    "serviceAddress": "http://localhost:5050/"
}
```

Parameters

Type	Name	Description	Schema
Path	**uuid** *required*	The UUID of the Service to update	string (uuid)
Body	**service** *optional*	The updated Service	Service

Responses

HTTP Code	Description	Schema
200	Success	No Content
404	Not Found	No Content

Consumes

- application/json-patch+json
- application/json
- text/json
- application/*+json

Tags

- Service

Delete a Service by UUID

```
DELETE /api/Service/{uuid}
```

Description

Example output:

```
DELETE /api/Services/{uuid}
{
    "uuid": "123e4567-e89b-12d3-a456-426655440000",
    "name": "name",
    "serviceType": 0,
    "isActive": true,
    "displayName": "Name",
    "serviceAddress": "http://localhost:5050/"
}
```

Parameters

Type	Name	Description	Schema
Path	**uuid** *required*	The UUID of the Service to delete	string (uuid)

Responses

HTTP Code	Description	Schema
200	Success	No Content
404	Not Found	No Content

Tags

- Service

Create Topology

```
POST /api/Topologies
```

Description

For more information on the Topology model, see documentation of Topology Service

Example Topology/output:

```
POST api/Topologies
{
    "topologyId": 0,
    "topologyName": "name",
    "isActive": true,
```

```
    "conveyorIds": [
        0, 1, 2, 3
    ],
    "conveyorLinkIds": {
        "0": {
            "0": 1
        },
        "1": {
            "0": 2
        },
        "2": {
            "0": 3
        },
        "3": {
            "0": 0
        }
    }
}
```

Parameters

Type	Name	Description	Schema
Body	**topologyJson** *optional*	The Topology	object

Responses

HTTP Code	Description	Schema
201	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Topologies

Get a shallow list of Topologies

```
GET /api/Topologies
```

Description

For more information on the Topology model, see documentation of Topology Service

Example output:

```
GET /api/Topologies
[
    {
        "topologyId": 0,
        "topologyName": "name",
        "isActive": true,
        "conveyorIds": null,
        "conveyorLinkIds": null
    },
    {
        "topologyId": 1,
        "topologyName": "name2",
        "isActive": false,
        "conveyorIds": null,
        "conveyorLinkIds": null
    }
]
```

Responses

HTTP Code	Description	Schema
200	Success	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Tags

- Topologies

Get a Topology by ID

```
GET /api/Topologies/{id}
```

Description

For more information on the Topology model, see documentation of Topology Service

Example output:

```
GET api/Topologies/{id}
{
    "topologyId": 0,
    "topologyName": "name",
    "isActive": true,
    "conveyorIds": [
        0, 1, 2, 3
    ],
    "conveyorLinkIds": {
        "0": {
            "0": 1
        },
        "1": {
            "0": 2
        },
        "2": {
            "0": 3
        },
        "3": {
            "0": 0
        }
    }
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	Topology ID	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Tags

- Topologies

Update Topology

```
PUT /api/Topologies/{id}
```

Description

For more information on the Topology model, see documentation of Topology Service

Example of Topology:

```
PUT api/Topologies/{id}
{
    "topologyId": 0,
    "topologyName": "name",
    "isActive": true,
    "conveyorIds": [
        0, 1, 2, 3
    ],
    "conveyorLinkIds": {
        "0": {
            "0": 1
        },
        "1": {
            "0": 2
        },
        "2": {
            "0": 3
        },
        "3": {
            "0": 0
        }
    }
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	ID of the Topology	integer (int32)
Body	**topologyJson** *optional*	The updated Topology	object

Responses

HTTP Code	Description	Schema
204	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Topologies

Delete Topology with ID

```
DELETE /api/Topologies/{id}
```

Description

For more information on the Topology model, see documentation of Topology Service

Example output:

```
DELETE api/Topologies/{id}
{
    "topologyId": 0,
    "topologyName": "name",
    "isActive": true,
    "conveyorIds": [
        0, 1, 2, 3
    ],
    "conveyorLinkIds": {
        "0": {
            "0": 1
        },
        "1": {
            "0": 2
```

```
        },
        "2": {
            "0": 3
        },
        "3": {
            "0": 0
        }
    }
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	ID of the Topology	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content
502	Server Error	No Content
503	No Topology Service	No Content

Tags

- Topologies

Definitions

Service

Name	Schema
displayName *optional*	string
isActive *optional*	boolean
name *required*	string

Name	Schema
serviceAddress *optional*	string
serviceType *optional*	integer (int32)
uuid *optional*	string (uuid)

APPENDIX C

Topology Service API Documentation

Topology Service API

Overview

Version information

Version : v1

Paths

Create a new Application

```
POST /api/Applications
```

Description

Also adds Operations, Ranges and Parameters

Example of application:

```
POST /api/Application
{
    "applicationId": 0,
    "applicationName": "name",
    "resourceId": 0,
    "supportedOperations": [
        {
            "applicationOperationId": 0,
            "operationId": 0,
            "operationName": "name",
            "supportedRanges": [
                {
                    "id": 0,
                    "name": "name",
                    "parameters": [
                        {
                            "id": 0,
                            "name": "name",
                            "minimum": 0,
                            "maximum": 0,
                            "minimumFunction": "{X}*2",
                            "maximumFunction": "{X}*4",
                            "defaultValue": 0
                        }
                    ]
                }
```

```
            ]
        }
    ]
}
```

Parameters

Type	Name	Description	Schema
Body	**application** *optional*	The Application	Application

Responses

HTTP Code	Description	Schema
201	Success	No Content
400	Bad Request	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Applications

Get a shallow list of Applications

```
GET /api/Applications
```

Description

To speed up response times, this response does include nested objects

Example output:

```
GET /api/Applications
[
    {
        "applicationId": 0,
        "applicationName": "name",
```

```
        "resourceId": 0,
        "supportedOperations": null
    }
]
```

Responses

HTTP Code	Description	Schema
200	Success	< Application > array

Produces

- `application/json`

Tags

- Applications

Get a deep copy of an Application

```
GET /api/Applications/{id}
```

Description

Example output:

```
GET /api/Application/{id}
{
    "applicationId": 0,
    "applicationName": "name",
    "resourceId": 0,
    "supportedOperations": [
        {
            "applicationOperationId": 0,
            "operationId": 0,
            "operationName": "name",
            "supportedRanges": [
                {
                    "id": 0,
                    "name": "name",
                    "parameters": [
                        {
                            "id": 0,
                            "name": "name",
                            "minimum": 0,
```

```
                            "maximum": 0,
                            "minimumFunction": "{X}*2",
                            "maximumFunction": "{X}*4",
                            "defaultValue": 0
                        }
                    ]
                }
            ]
        }
    ]
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the Application to get	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	No Application with the ID	No Content

Tags

- Applications

Update an Application

```
PUT /api/Applications/{id}
```

Description

Also adds/updates/deletes Operations, Ranges and Parameters

Example of application:

```
PUT /api/Application/{id}
{
    "applicationId": 0,
    "applicationName": "name",
```

```
    "resourceId": 0,
    "supportedOperations": [
        {
            "applicationOperationId": 0,
            "operationId": 0,
            "operationName": "name",
            "supportedRanges": [
                {
                    "id": 0,
                    "name": "name",
                    "parameters": [
                        {
                            "id": 0,
                            "name": "name",
                            "minimum": 0,
                            "maximum": 0,
                            "minimumFunction": "{X}*2",
                            "maximumFunction": "{X}*4",
                            "defaultValue": 0
                        }
                    ]
                }
            ]
        }
    ]
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	A valid Application ID	integer (int32)
Body	**application** *optional*	The updated Application	Application

Responses

HTTP Code	Description	Schema
204	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Consumes

- `application/json-patch+json`

- application/json
- text/json
- application/*+json

Tags

- Applications

Delete the application with id

```
DELETE /api/Applications/{id}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the application to delete	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Tags

- Applications

Create a module conveyor

```
POST /api/ModuleConveyors
```

Description

Example of module conveyor:

```
POST /api/ModuleConveyors
{
    "moduleConveyorId": 0,
    "moduleConveyorName": "unique-name",
    "plcIpAddressString": "10.0.0.144",
```

```
    "plcMacAddressString": "0011223344",
    "resourceId": 0
}
```

Parameters

Type	Name	Description	Schema
Body	**moduleConveyor** *optional*	The new module conveyor	ModuleConveyor

Responses

HTTP Code	Description	Schema
201	Success	No Content
400	Bad Request	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- ModuleConveyors

Get all module conveyors

```
GET /api/ModuleConveyors
```

Description

Use GET api/ModuleConveyors/id to retrieve a deep copy of the module conveyor

Example output:

```
GET /api/ModuleConveyors
[
    {
        "moduleConveyorId": 0,
        "moduleConveyorName": "unique-name",
        "plcIpAddressString": "10.0.0.144",
```

```
        "plcMacAddressString": "0011223344",
        "resourceId": 0
    },
    {
        "moduleConveyorId": 1,
        "moduleConveyorName": "special-name",
        "plcIpAddressString": "10.0.0.145",
        "plcMacAddressString": "0011223355",
        "resourceId": 1
    }
]
```

Responses

HTTP Code	Description	Schema
200	Success	< ModuleConveyor > array

Produces

- `application/json`

Tags

- ModuleConveyors

Get a module conveyor by ID

```
GET /api/ModuleConveyors/{id}
```

Description

Example output:

```
GET /api/ModuleConveyors/{id}
{
    "moduleConveyorId": 0,
    "moduleConveyorName": "unique-name",
    "plcIpAddressString": "10.0.0.144",
    "plcMacAddressString": "0011223344",
    "resourceId": 0
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the module conveyor	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Tags

- ModuleConveyors

Update a ModuleConveyor

```
PUT /api/ModuleConveyors/{id}
```

Description

Example of module conveyor:

```
POST /api/ModuleConveyors/{id}
{
    "moduleConveyorId": 0,
    "moduleConveyorName": "unique-name",
    "plcIpAddressString": "10.0.0.144",
    "plcMacAddressString": "0011223344",
    "resourceId": 0
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	A valid ID of the ModuleConveyor	integer (int32)
Body	**moduleConveyor** *optional*	The ModuleConveyor	ModuleConveyor

Responses

HTTP Code	Description	Schema
204	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- ModuleConveyors

Delete a ModuleConveyor

```
DELETE /api/ModuleConveyors/{id}
```

Description

Example output:

```
DELETE /api/ModuleConveyors/{id}
{
    "moduleConveyorId": 0,
    "moduleConveyorName": "unique-name",
    "plcIpAddressString": "10.0.0.144",
    "plcMacAddressString": "0011223344",
    "resourceId": 0
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	A valid ID of the ModuleConveyor to delete	integer (int32)

Responses

HTTP Code	Description	Schema

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Tags

- ModuleConveyors

Create a Resource

```
POST /api/Resources
```

Description

Example output:

```
POST /api/Resources
{
    "applicationId": 0,
    "moduleConveyorId": 0,
    "resourceId": 0,
    "resourceName": "Name"
}
```

Parameters

Type	Name	Description	Schema
Body	**resource** *optional*	The Resource to create	Resource

Responses

HTTP Code	Description	Schema
201	Success	No Content
400	Bad Request	No Content

Consumes

- `application/json-patch+json`
- `application/json`

- text/json
- application/*+json

Tags

- Resources

Get a list of all Resources

```
GET /api/Resources
```

Description

Example output:

```
GET /api/Resources
[
    {
        "applicationId": 0,
        "moduleConveyorId": 0,
        "resourceId": 0,
        "resourceName": "Name"
    },
    {
        "applicationId": 1,
        "moduleConveyorId": 2,
        "resourceId": 1,
        "resourceName": "Name"
    }
]
```

Responses

HTTP Code	Description	Schema
200	Success	< Resource > array

Produces

- application/json

Tags

- Resources

Get a Resource by ID

```
GET /api/Resources/{id}
```

Description

Example output:

```
GET /api/Resources/{id}
{
    "applicationId": 0,
    "moduleConveyorId": 0,
    "resourceId": 0,
    "resourceName": "Name"
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The Resource ID	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Tags

- Resources

Update a Resource

```
PUT /api/Resources/{id}
```

Description

For more information about the Resource model, see the Topology Service documentation

Example output:

```
PUT /api/Resources/{id}
{
    "applicationId": 0,
    "moduleConveyorId": 0,
    "resourceId": 0,
    "resourceName": "Name"
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the Resource to update	integer (int32)
Body	**resource** *optional*	The updated Resource	Resource

Responses

HTTP Code	Description	Schema
204	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Resources

Delete a Resource

```
DELETE /api/Resources/{id}
```

Description

Example output:

```
DELETE /api/Resources/{id}
{
    "applicationId": 0,
    "moduleConveyorId": 0,
    "resourceId": 0,
    "resourceName": "Name"
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the Resource to delete	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Tags

- Resources

Create Topology

```
POST /api/Topologies
```

Description

Example Topology/output:

```
POST api/Topologies
{
    "topologyId": 0,
    "topologyName": "name",
    "isActive": true,
    "conveyorIds": [
        0, 1, 2, 3
    ],
    "conveyorLinkIds": {
```

```
            "0": {
                "0": 1
            },
            "1": {
                "0": 2
            },
            "2": {
                "0": 3
            },
            "3": {
                "0": 0
            }
        }
    }
```

Parameters

Type	Name	Description	Schema
Body	**topology** *optional*	The Topology	Topology

Responses

HTTP Code	Description	Schema
201	Success	No Content
400	Bad Request	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Topologies

Get a shallow list of Topologies

```
GET /api/Topologies
```

Description

To speed up response times, this response does not include nested objects or references

Example output:

```
GET /api/Topologies
[
    {
        "topologyId": 0,
        "topologyName": "name",
        "isActive": true,
        "conveyorIds": null,
        "conveyorLinkIds": null
    },
    {
        "topologyId": 1,
        "topologyName": "name2",
        "isActive": false,
        "conveyorIds": null,
        "conveyorLinkIds": null
    }
]
```

Responses

HTTP Code	Description	Schema
200	Success	< Topology > array

Produces

- `application/json`

Tags

- Topologies

Get a Topology by ID

```
GET /api/Topologies/{id}
```

Description

Example output:

```
GET api/Topologies/{id}
{
```

```
    "topologyId": 0,
    "topologyName": "name",
    "isActive": true,
    "conveyorIds": [
        0, 1, 2, 3
    ],
    "conveyorLinkIds": {
        "0": {
            "0": 1
        },
        "1": {
            "0": 2
        },
        "2": {
            "0": 3
        },
        "3": {
            "0": 0
        }
    }
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	Topology ID	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Tags

- Topologies

Update Topology

```
PUT /api/Topologies/{id}
```

Description

Example of Topology:

```
PUT api/Topologies/{id}
{
    "topologyId": 0,
    "topologyName": "name",
    "isActive": true,
    "conveyorIds": [
        0, 1, 2, 3
    ],
    "conveyorLinkIds": {
        "0": {
            "0": 1
        },
        "1": {
            "0": 2
        },
        "2": {
            "0": 3
        },
        "3": {
            "0": 0
        }
    }
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	ID of the Topology	integer (int32)
Body	**topology** *optional*	The updated Topology	Topology

Responses

HTTP Code	Description	Schema
204	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Consumes

- `application/json-patch+json`

- application/json
- text/json
- application/*+json

Tags

- Topologies

Delete Topology with ID

```
DELETE /api/Topologies/{id}
```

Description

Example output:

```
DELETE api/Topologies/{id}
{
    "topologyId": 0,
    "topologyName": "name",
    "isActive": true,
    "conveyorIds": [
        0, 1, 2, 3
    ],
    "conveyorLinkIds": {
        "0": {
            "0": 1
        },
        "1": {
            "0": 2
        },
        "2": {
            "0": 3
        },
        "3": {
            "0": 0
        }
    }
}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	ID of the Topology	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad Request	No Content
404	Not Found	No Content

Tags

- Topologies

Definitions

Application

Name	Schema
applicationId *optional*	integer (int32)
applicationName *optional*	string
resourceId *optional*	integer (int32)
supportedOperations *optional*	< ApplicationOperation > array

ApplicationOperation

Name	Schema
applicationOperationId *optional*	integer (int32)
operationId *optional*	integer (int32)
operationName *optional*	string
supportedRanges *optional*	< Range > array

ModuleConveyor

Name	Schema

Name	Schema
moduleConveyorId *optional*	integer (int32)
moduleConveyorName *optional*	string
plcIpAddressString *optional*	string
plcMacAddressString *optional*	string
resourceId *optional*	integer (int32)

Range

Name	Schema
id *optional*	integer (int32)
name *optional*	string
parameters *optional*	< RangeParameter > array

RangeParameter

Name	Schema
defaultValue *optional*	number (double)
id *optional*	integer (int32)
maximum *optional*	number (double)
maximumFunction *optional*	string
minimum *optional*	number (double)
minimumFunction *optional*	string

Name	Schema
name *optional*	string

Resource

Name	Schema
applicationId *optional*	integer (int32)
moduleConveyorId *optional*	integer (int32)
resourceId *optional*	integer (int32)
resourceName *optional*	string

Topology

Name	Schema
conveyorIds *optional*	< integer (int32) > array
conveyorLinkIds *optional*	< string, < string, integer (int32) > map > map
isActive *optional*	boolean
topologyId *optional*	integer (int32)
topologyName *optional*	string

APPENDIX D

Order Service API Documentation

Order Service API

Overview

Version information

Version : v1

Paths

Creates an order position

```
POST /api/OrderPositions
```

Parameters

Type	Name	Schema
Body	**orderPosition** *optional*	OrderPosition

Responses

HTTP Code	Description	Schema
200	Success	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- OrderPositions

Gets all the Order positions

```
GET /api/OrderPositions
```

Responses

HTTP Code	Description	Schema
200	Success	< OrderPosition > array

Produces

- `application/json`

Tags

- OrderPositions

Gets all the order positions of an order

```
GET /api/OrderPositions/Order/{id}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the parent order	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content

Tags

- OrderPositions

Gets an individual order position

```
GET /api/OrderPositions/{id}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the order position	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content

Tags

- OrderPositions

Deletes an order position

```
DELETE /api/OrderPositions/{id}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the order position to be deleted	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content

Tags

- OrderPositions

Adds a new order

```
POST /api/Orders
```

Parameters

Type	Name	Schema
Body	**order** *optional*	Order

Responses

HTTP Code	Description	Schema
200	Success	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Orders

Gets all the existing orders

```
GET /api/Orders
```

Responses

HTTP Code	Description	Schema
200	Success	< Order > array

Produces

- `application/json`

Tags

- Orders

Gets individual orders

```
GET /api/Orders/{id}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the order	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content

Tags

- Orders

Updates the fields of an order

```
PUT /api/Orders/{id}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the order to be updated	integer (int32)
Body	**order** *optional*		Order

Responses

HTTP Code	Description	Schema
200	Success	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Orders

Deletes an order

```
DELETE /api/Orders/{id}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the order to be deleted	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content

Tags

- Orders

Not implemented yet

```
POST /api/Products
```

Parameters

Type	Name	Schema
Body	**value** *optional*	string

Responses

HTTP Code	Description	Schema
200	Success	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Products

Gets all the products

```
GET /api/Products
```

Responses

HTTP Code	Description	Schema

HTTP Code	Description	Schema
200	Success	< Product > array

Produces

- `application/json`

Tags

- Products

Gets an individual product

```
GET /api/Products/{id}
```

Parameters

Type	Name	Description	Schema
Path	**id** *required*	The ID of the product	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content

Tags

- Products

Not implemented yet

```
PUT /api/Products/{id}
```

Parameters

Type	Name	Schema
Path	**id** *required*	integer (int32)
Body	**value** *optional*	string

Responses

HTTP Code	Description	Schema
200	Success	No Content

Consumes

- `application/json-patch+json`
- `application/json`
- `text/json`
- `application/*+json`

Tags

- Products

Not implemented yet

```
DELETE /api/Products/{id}
```

Parameters

Type	Name	Schema
Path	**id** *required*	integer (int32)

Responses

HTTP Code	Description	Schema
200	Success	No Content

Tags

- Products

Definitions

Order

Name	Schema
endTime *optional*	string (date-time)

Name	Schema
name *required*	string
orderId *optional*	integer (int32)
orderPositions *optional*	< OrderPosition > array
plannedStartTime *optional*	string (date-time)
priority *optional*	enum (Low, Medium, High)
startTime *optional*	string (date-time)
state *optional*	enum (Created, Started, Completed)

OrderPosition

Name	Schema
endTime *optional*	string (date-time)
order *optional*	Order
orderPositionId *optional*	integer (int32)
plannedStartTime *optional*	string (date-time)
product *optional*	Product
startTime *optional*	string (date-time)
state *optional*	enum (Created, Started, Completed)
workPlanNumber *optional*	integer (int32)

Product

Name	Schema
description *optional*	string
orderPosition *optional*	OrderPosition
partNumber *optional*	integer (int32)
productId *optional*	integer (int32)
workPlanNumber *optional*	integer (int32)