

**FACULTY  
OF MATHEMATICS  
AND PHYSICS**  
Charles University

**BACHELOR THESIS**

Hana Nekvindová

**MetaRMS – information systems  
building platform**

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D.

Study programme: Computer Science

Study branch: Programming and Software systems

Prague 2019



I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author



Title: MetaRMS – information systems building platform

Author: Hana Nekvindová

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Data are an essential element of the present world. The problem of storing data concerns everybody, from a large company with information about their clients to individual users with their shopping lists. Options vary between a simple Excel sheet and expensive custom solution. A general software solution to cover these cases is needed. However, the requirements on the structure of the data differ for every use-case.

This thesis aims to solve this problem by creating an application generating software. The software generates a custom application when provided with the description of the data structure. For that, we define the format of the description of the data structure and analyze various approaches to the implementation of the application generating software.

Our solution contains an ASP.NET Core server application and an example web client application communicating over the public JSON API. The server accepts the description and creates an application accordingly. The solution also contains a library, that is used by the example web client and is reusable by other client front-ends.

Keywords: ASP.NET Core multiplatform API client-server generate application



I would like to express my gratitude to Mgr. Pavel Ježek, Ph.D., the supervisor of this thesis for his time, guidance, feedback and all the advice he has provided. Also, a special thanks go to my family and Pavel.



# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Basic requirements and limitations . . . . .	4
1.2	Naming conventions . . . . .	7
1.3	Requirements . . . . .	8
1.4	Existing tools . . . . .	10
1.5	Goals . . . . .	13
<b>2</b>	<b>Workflow and application descriptor</b>	<b>15</b>
2.1	Basic application information . . . . .	16
2.2	Datasets . . . . .	16
2.3	Attributes . . . . .	19
2.3.1	Complete list of available attribute properties . . . . .	21
2.4	Application descriptor structure . . . . .	31
2.5	Application descriptor validation . . . . .	31
2.5.1	Necessary validations . . . . .	32
2.6	Summary and next steps . . . . .	34
<b>3</b>	<b>Implementation analysis</b>	<b>35</b>
3.1	Software architecture . . . . .	35
3.2	Language and environment . . . . .	38
3.3	References . . . . .	40
3.4	Authentication . . . . .	42
3.5	Authorization . . . . .	45
3.6	Messages handling . . . . .	47
3.7	Application descriptor format . . . . .	48
3.7.1	Application descriptor validation . . . . .	50
3.7.2	Default values setting . . . . .	51
3.7.3	Final application descriptor structure in JSON . . . . .	51
3.8	Data layer . . . . .	53
3.8.1	Database schema . . . . .	53
3.8.2	Data storage choice . . . . .	58
3.8.3	Accessing the database from the code . . . . .	59
3.9	Hosting . . . . .	60
<b>4</b>	<b>Implementation</b>	<b>63</b>
4.1	Basic workflow and requirements . . . . .	63
4.2	MetaRMS processes . . . . .	64
4.2.1	Application initialization . . . . .	65

4.2.2	Authentication . . . . .	66
4.2.3	Operations . . . . .	67
4.2.4	Settings . . . . .	73
4.3	Solution structure . . . . .	74
4.3.1	Core . . . . .	74
4.3.2	SharedLibrary . . . . .	77
4.4	Issues . . . . .	79
4.4.1	HTTPS support . . . . .	79
4.4.2	Sending emails . . . . .	80
4.4.3	Cultural info . . . . .	81
4.4.4	Login length on production . . . . .	81
<b>5</b>	<b>User guide</b>	<b>83</b>
5.1	Client application developer's guide . . . . .	83
5.2	System administrator's guide . . . . .	84
5.3	Application administrator's guide . . . . .	87
5.4	End-user's guide . . . . .	95
<b>6</b>	<b>Conclusion</b>	<b>99</b>
6.1	Future work . . . . .	101
	<b>Attachments</b>	<b>107</b>

# 1. Introduction

Nowadays people are living in a world in which the data are more important than ever. Whether you are a small startup or a corporate company, you need to store your data somewhere safe and easy to access. But not only that, even regular people need to store some data, be it a ToDo list, phone numbers or friends' birthdays. The most popular and most commonly used storages these days are relational databases. Relational databases and their inner representations might differ, but in most cases, the main principle is always the same—large interconnected tables containing data of various types.

End users need not only to store the data, but they also need to retrieve the data in an easy way, preferably without using SQL queries. Common users will probably require a user-friendly interface, that allows them to (1.) create, (2.) read, (3.) update and (4.) delete the data—this is known as performing CRUD (abbreviation for operations 1. – 4.) operations on the data.

If a person or a company wants to use a database as data storage and retrieve the data via CRUD operations a problem emerges because everybody needs at least slightly different database model to suit their exact needs. For example, we can have an e-shop. An e-shop needs to store information about its clients as well as employees, their products, all the orders, their statuses and much more other crucial information for their business. Compared to that if a regular person wants to store a ToDo list, there will be a name of the task to be done and an indicator if the task is done. Apart from that, one person would like to store a priority with each of the tasks, another one would like to add deadlines and the next one a person responsible for the task. As we can see, even with storing two ToDo lists, the data structures can be very different, which means we can not even have templates for most frequently used cases to fit all the users and their requirements.

As demonstrated, needs for data storage differ significantly for each use case. For a company, a typical scenario is to have the database tailored to the needs of the company as well as the user interface by a team of developers. This way the company gets exactly what it is looking for. However, getting a custom solution may not be accessible for small companies with a tight budget or even for individual users who just want to store their shopping list for the next week.

## **Original idea**

One of such small companies has reached us with a request for custom software for their business. Their goal was to have an application for administration of everyday business tasks. This is a great example of a company that needs to represent their inner structure in software but have not found such one fitting their needs and budget.

The requested solution should:

- Be able to store past, current and future orders with custom fields and additional information about their customers.
- Have a user administration managed by the authorized person.
- Keep the data save not only against hackers from the outside, but also make sure that no one from the inside could access data that the person was not allowed to see.

This is where the original idea for this thesis comes from. After market research, we have found that there might be many such companies or individuals looking for a custom made software for data administration. We have thus come with a more generalized idea to create a system to satisfy the needs of this whole target group.

## **1.1 Basic requirements and limitations**

In the introduction, the main problem of data storing and retrieving based on company structure was explained with an idea to create a universal software, that will be general enough to store data and provide CRUD operations on them for any data structure and so to suit any company's or individual's needs, defined by the administrator. In the following subsection, we are going to describe some representative scenarios in more detail.

### **Representative scenarios**

Before listing all of the requirements on the software, we present several carefully chosen usage examples to show how can various structures of data differ. The structures are divided into three groups by the structure size and complexity.

1. Department of a larger company:

- (a) **An applicant tracking system (ATS).** *It is a software application that enables the electronic handling of recruitment needs* (cited from Wikipedia [App19]). Such application stores jobs in the company with all the necessary information such as name, requirements or salary. It also stores candidates for positions with their name, education, and other important details. Finally, there are hiring processes that connect jobs with individual candidates. Such application has users divided into user groups such as recruiters, hiring managers or system administrators. Advanced ATS systems allow integration with job portal advertising, career pages or other company systems.
- (b) **Inventory management software.** *It is a software system for tracking inventory levels, orders, sales, and deliveries* (cited from Wikipedia [Inv19]). In this type of software, the key items are products. They are stored with their barcode, description, location, the amount left and other information. Also, old and new orders, manufacturers and for companies with more warehouses, also individual warehouses are stored. There are users with different authorizations, that need to use such system—from warehouseman to warehouse manager. Advance inventory management software can send emails with orders, offer integration with barcode scanners and functions for inventory optimization.

## 2. Small company:

- (a) **A municipal library.** Software for storing books and authors data, readers with access to the library and their borrowings. If the library is using a single system for all internal processes, then also for example payrolls needs to be stored here. This creates a diverse group of the software users, be it librarians, library support members, accountants and library managers. Advanced systems offer barcode scanner integration and email sending service to inform readers about a book return date.
- (b) **Package delivery company,** that needs to keep a system in their branches, transport facilities and keep order in the packages—its statuses, delivery dates, current locations, and target addresses. Employees with different competencies can access different data within the application. More advanced systems can track the GPS location of individual packages or send SMS about delivery times to the customers.

3. Individual user:

- (a) **A ToDo list** with custom fields. As mentioned at the beginning of this chapter, requirements for ToDo lists can differ between users—some of them prefer to store deadlines, others want to store links to more information. Because of that, no ToDo application can fit all users and this is where custom made applications can excel. The main advantage of ToDo applications with predefined fields is, that they can perform data analytics over the data and show for example graphs with numbers of tasks done per week.
- (b) **Sports tracker.** It is an application where users can record their performance in sports, for example, how many kilometers they ran, how much weight they lifted or how many pushups they did. They can record their weight, what did they eat during the day or how did they feel after the workout. In case of a personal trainer as a second user of the application, the trainer can provide feedback to the user. Advanced sports trackers include integration with smartwatches or some other health applications.

From these examples, it follows that the key requirement for fulfilling all of the above-mentioned scenarios is the ability to describe an arbitrary data structure, based on the company structure or personal needs. In this description, only the basic structure was presented. Each of the example applications can be divided into database tables and columns with defined value types, be it basic data types (text, number, date, etc.) or references to other data.

On the other hand, each of the application descriptions contains also part describing an advanced version of the application. That section contains features highly dependent on the application structure and other company or individual's software and hardware. It is important to state, that the goal of this thesis is not to fulfill such specific requirements, as:

- integration with other systems (job portal advertising, career pages or other company systems, GPS tracking systems, health applications, calendar, etc.);
- send email or SMS;
- integration with hardware (barcode scanners, smartwatches, etc.);
- data processing functions (generating graphs, data analytics, etc.).

## Common solution approach

If we want to satisfy the basic needs of all the companies/users from the examples above—to have an application with custom fields based on own structure—we need to have a software that can, based on some description, generate a custom application and allow access to it to users with different competencies. The application descriptor has to provide a way to describe the application structure in the form of database tables with columns of basic data types or references within the application.

## 1.2 Naming conventions

At the end of the previous section, some not yet defined terms appeared. Before going any deeper into the problem, it is time to define them and some other frequently used terms, that will be used in the following text consistently:

**Application** A piece of individual software, that can be described by some kind of application descriptor. Users can log in and perform CRUD operations with data they have rights to access.

**Business application** An application, that reflects company structure and which targets company employees. Specifically, it means that applications for customers (such as e-shops, informative web pages or mobile applications with information for customers, such as opening times) do not meet this definition.

**Application descriptor** A text/file containing all the necessary information for an application generating software to generate an application.

**Application generating software** A type of software that can generate an application. The ability to generate means that for creating a new application no programming skills are required and after providing a description of what the result should be, the application is created.

**Dataset** A collection of data, that can correspond to a database table. Such a dataset can contain an arbitrary number of dataset attributes, described in more details in the following bullet.

**Dataset attribute** A single one item of the dataset, which corresponds to a database column. Attributes can have many properties, which are described in the following bullet.

**Attribute properties** An individual piece of information about dataset's attribute, such as its name, description, type or if it is required. An attribute can have various properties and it is up to a discussion which of them are crucial and it will be elaborated in more detail in subsection 2.3.1.

## 1.3 Requirements

Since we have described a problem and defined a terminology, this section will get deeper into the problem by listing and discussing requirements on the application generating software in more details. For each requirement, we will first describe reasons for its establishment and after that, we will name it.

Back at the end of section 1.1, we have marked the most important feature of an application generating software as the ability to take a description of an application and create a real application based on it. This requirement is crucial and the following text will refer to it as the requirement:

**R1 Application generating software is required to be able to take a definition of various structures and thus create different applications.**

As seen from the market shares on [Ope18], the reason for targeting multiple platforms is obvious. This requirement can be divided into two subsections since mobile devices cannot be overlooked. Obvious from the market shares, mentioned above, the Windows operating system is mostly used on desktops and laptops, followed by Apple's macOS. On mobile devices, there are two main platform choices—Android and iOS. To reach the largest amount of users, it is important to target all of the above-listed platforms. This gives us the requirement:

**R2 Application generating software is required to be multiplatform.**

Targeting multiple platforms is useless if the generated result application is too difficult to use by regular users. As we have explained in the Representative scenarios subsubsection of section 1.1, the application end users can be at different computer knowledge levels, thus:

**R3 Applications generated by the application generating software must be easy to use for end users.**

On the other hand, the process of creating a new application is not targeted at the regular users, so this part might contain some advanced knowledge from the IT world, even though it should still remain easy and quick, so:

**R4 Application generating software should provide an easy way to create a new application.**

After providing the description of the application, companies or individuals should not wait too long for the final application. The time between description providing and the start of using the application should be minimized. Because of this:

**R5 Time delta between creating a new application and its using by the end-users should be minimized for applications created by the application generating software.**

As mentioned in the Original idea subsubsection in chapter 1, for the company, that has reached us, one of the key requirements is the safety of their data. This results in two requirements. First of them is authentication, to ensure no one can access data without first logging into the application as an authenticated user:

**R6 Applications generated by the application generating software must contain authentication.**

The second one is a requirement for authorization. Without this requirement, administrators could not divide users into groups according to their rights. That would disrupt company structure because users would be able to access data beyond their rights:

**R7 Applications generated by the application generating software must contain authorization.**

To make application usage convenient for non-English speaking users, it is important to include a possibility to switch the user interface to various languages, thus:

**R8 Applications generated by the application generating software should have multilingual support.**

There are also some properties of the application generating software that do not directly affect a regular user. The software may not support all the major platforms mentioned in requirement R2 (1.3), but there would be an easy solution via public API. This way anybody could create their own

client application connected to the original one. A similar requirement is an easily extendable source code. This feature will allow programmers to extend existing software to other platforms. This together gives us two more requirements:

**R9 Application generating software should have an API so anyone can use it as an endpoint.**

**R10 Application generating software should have an easily extendable source code.**

With this complete list of requirements, we also need to note some points that are not required by the application generating software.

- Work offline – applications generated by the application generating software are not required to work offline. This feature is not required, because business applications are mostly used in office environments with a stable internet connection or in the field in the cities. The requirement for the internet connection also allows data to be synchronized all the time instantly.
- Target everyone – as already mentioned in chapter 1 rather than targeting huge international commercial businesses, we want to focus on smaller companies and individuals.
- Offer features highly dependant on the application structure – as stated in the Representative scenarios subsubsection of section 1.1 the goal is not to offer features, that depend on company or individual's hardware and software gear. We will rather focus on a general solution.

## 1.4 Existing tools

With the specific list of requirements from section 1.3, we have done market research. The research was made with a goal to map the market of the already existing application generating software.

The g2.com website proved to be a great source of information about various application generating software. We were interested in two of the categories, Low-Code Development Platforms Software [G2l] and No-Code Development Platforms Software [G2n]. In the following subsections, we will briefly go through both of these categories and we will highlight key features of software belonging into them.

## **Low-Code Development Platforms Software**

As defined by G2 on [G2l]: *Low-code development platforms provide development environments that allow businesses to develop software quickly with minimal coding, minimizing the need for extensive coding experience. The platforms provide base-level code, scripts, and integrations so companies can prototype, build, or scale applications without developing complex infrastructures. Both developers and non-developers can use these tools to practice rapid application development with customized workflows and functionality.*

From their highest rated software, we will briefly introduce the two most popular.

**OutSystems** This Low-code Development Platform offers drag-and-drop editor, that users download and then can assemble their mobile and web applications in it. This editor allows to define a data model, business logic, workflow processes, and user interfaces and also allows to enhance some of the parts with code. The main advantage this platform offers is the possibility of integrating existing systems and real-time performance monitoring. On the other hand, mobile application generated by OutSystems must be placed to mobile application stores by the user, which adds additional costs. A free edition is suited to learn how to build and deploy applications and is suitable for small applications. For larger companies, OutSystems offers paid subscriptions. Additional information can be found on [www.outsystems.com](http://www.outsystems.com).

**FileMaker** This development platform lies between Low-code and No-code platforms and is more suitable for companies that are already storing data another way, for example in Excel files. FileMaker allows users to import their data and then create different views over them. The disadvantage of FileMaker is, that free demo is only for 45 days and after that, a paid license needs to be bought. More information can be found on [www.filemaker.com](http://www.filemaker.com).

## **No-Code Development Platforms Software**

By G2 definition on [G2n]: *No-code development platforms provide drag-and-drop tools that enable businesses to develop software quickly without coding. The platforms provide WYSIWYG editors and drag-and-drop components to quickly assemble and design applications. Both developers and non-developers*

*can use these tools to practice rapid application development with customized workflows and functionality.*

From all the existing tools we will briefly mention just the most popular.

**AppSheet** This No-code Development Platform uses already existing spreadsheet data stored online on services like Google Sheets, Microsoft Excel on Office 365, Microsoft Excel on Dropbox, etc. These spreadsheets are accessed to display data in generated mobile or web application. Any changes in the data are synchronized and written back to the spreadsheet. An important feature is that all the data are owned by the application creator and not by the AppSheet company. This platform is free for personal use with a single user and offers paid subscriptions for businesses. More information can be found on [www.appsheet.com](http://www.appsheet.com).

**FileMaker** Already mentioned in the Low-Code Development Platforms Software subsubsection of this subsection 1.4.

**Airtable** This No-code Development Platform uses advantages of both spreadsheets and databases to store data and allow different views over it. A new application can be created from scratch or use one of their templates. When the application is created it can be shared between application users and the data can be embedded to a webpage or accessed via API. Airtable is free to use for teams of any size and also offers paid subscriptions. More about Airtable can be found on [airtable.com](http://airtable.com).

## Summary

While doing the market research, we have noticed that the application's logic is mostly based on a spreadsheets logic. The advantage of this approach is that users already know how spreadsheets work. All of the elaborated platforms provide an API to access the data and on top of that when using AppSheet the data are actually stored on the creator's online storage, which makes accessing them even easier. An interesting part of the platforms is the drag-and-drop interface, which provides an easy way to define the application for non-programmers.

Even though some of the elaborated platforms were close to what we expect from the software based on the requirements from section 1.3, there will be some significant differences in our approach. The most important is, that we will provide means to have both the software and the data under the control of the administrator. Also if we stick to the requirement R10 (1.3)

the source code of the software would be easily extendable and will provide more flexibility in case of modifications in the future.

## 1.5 Goals

With all the information we found out so far and equipped with the right terminology from section 1.2, the thesis topic can be now defined. The goal of this thesis is to develop an own application generating software, that will satisfy all groups from Representative scenarios subsubsection of section 1.1 and fulfill the requirements from section 1.3.



## 2. Workflow and application descriptor

Since this chapter will contain terms *application*, *application generating software*, *application descriptor* and other from naming conventions defined in section 1.2 we will strictly write these special terms in italics, just like at the beginning of this sentence.

With the knowledge of our goal, we need to think about the workflow of the creation of a new *application* by a user. Since we want the *application* to have a custom structure, we need to describe this structure somehow. This description must have a specific format, because *application generating software* (defined in section 1.2) needs to understand it. The description of a new *application* is called *application descriptor* (as defined in section 1.2). With *application descriptor* an *application generating software* can create the *application* for end users. This workflow is displayed in the following figure 2.1.

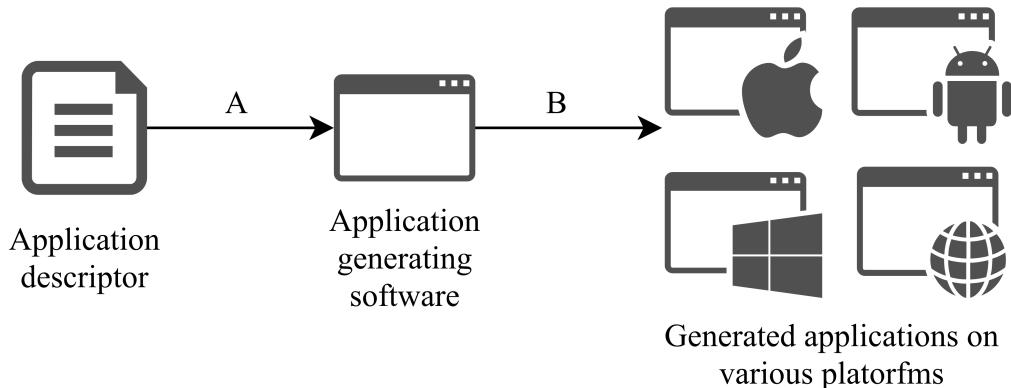


Figure 2.1: *Application* creation workflow.

There are two arrows in the firure 2.1, except of the above mentioned *application descriptor*, *application generating software* and the final generated *applications*. Arrow A, connecting *application descriptor* and *application generating software* expresses the process of providing the *application descriptor* to the *application generating software*. Because of it the *application generating software* can create the *application* based on the *application descriptor* provided. The arow B from the *application generating software* to the final generated *applications* shows the process of *application* generation.

As we can see, the *application descriptor* is the main component on which all the functionality of the *application generating software* depends. The *application descriptor* is required to be able to describe an arbitrary *application* as stated in the requirement *R1* (1.3). By the requirement *R4* (1.3) it must provide an easy way to define the *application* so users with basic IT knowledge are able to define their *application*.

In the following sections we will elaborate individual parts of the *application descriptor* and reasons for including them.

## 2.1 Basic application information

Since the *application generating software* needs to hold many *application descriptors*, it is crucial that we are able to always find the correct *application descriptor* and generate the right *application* for each user. Because of this, each *application* needs to have a **unique identifier**, that will distinguish the *application* from others. On the other hand, each *application* should have a **name**, that will be displayed so logged users know what *application* are they logged in. This name does not have to be unique since for example several users may like to call their *application* *My ToDo list*. As stated in the requirement *R8* (1.3) we want the *application generating software* to support various languages. With this in mind, we have incorporated a **default application language** into the *application descriptor*, even though only English will be supported in the release due to limited resources. Each *application* also needs to contain **datasets** elaborated in more details in the following section 2.2.

## 2.2 Datasets

As stated in the previous section 2.1 every *application* needs to contain *datasets*. Definition of the *application dataset* can be found in the section 1.2. Even though the structure of the *applications* will differ, all of them are required to have users, who will sign into and perform the CRUD operations on the *application* data. The information about *application* users can be represented in a *dataset*. Because of this, there will be two types of *datasets* in every *application*.

**User-defined datasets** Arbitrary *datasets* defined by the creator of the *application descriptor*. Since there is no use of *application* without a *dataset*—from its definition in section 1.2 it can be compared to a database without any tables, at least one *User-defined dataset* is

required. These *datasets* will be *application* specific—for representative scenarios mentioned in the Representative scenarios subsubsection of section 1.1 we can have following *datasets*:

- ATS (applicant tracking system) from scenario 1a – candidates, jobs, interviews, etc.
- Municipal library from scenario 2a – books, readers, borrowings, etc.
- Sports tracker from scenario 3b – types of sports, activities, etc.

**System datasets** *System datasets* are *datasets* used for the *application configuration*. They contain required as well as optional *dataset attributes*. The only currently supported *System dataset* is the *System users dataset*:

**System users dataset** *System dataset* containing the information about users of the *application*. This *dataset* will be required to be present in every *application descriptor* and it will hold an *application* specific configuration of the *application* users data. This *dataset* will contain settings of username and password and it will also additionally describe the structure of the users. Any additional information, such as the position within the company, number of kids or user's boss can be set in this *dataset* in a form of *dataset attributes* as defined in section 1.2. Due to the presence of the settings and additional information about users, creator of the *application* is required to specify this *dataset* in the *application descriptor* as it can not be generated automatically.

With this in mind we have to choose the representation of *datasets* in the *application descriptor*. We have thought of the following approaches:

1. Treat the *System datasets* the same way as the *User-defined datasets*. This way all the *datasets* would be equal and we would need to think about a way how to distinguish them and find the *System datasets* if necessary. With this approach users would need to understand only one way of defining a *dataset* and would not need to distinguish between writing the *System datasets* and the *User-defined datasets*.
2. Separate *System datasets* and *User-defined datasets*. With this approach the *System datasets* could have a slightly different structure from the *User-defined datasets*. For example, in the *System users dataset*, the password and the username can be expressed in a more

clear way. It would also be more straightforward for the author of the *application descriptor* to distinguish between the *System datasets* and the *User-defined datasets*.

From the approaches listed above, we have opted for the approach 2. The *System users dataset* will contain both the username and the password, which need to be treated in a different way than the rest of the *attributes*. Because of this and the fact, that the *System users dataset* is required in every *application descriptor*, we want to treat the *dataset* in a different way than the *User-defined datasets* and this separation allows us to do so.

Both the *User-defined datasets* and the *System datasets* need to have a **unique name** within the *application* to distinguish between the *datasets* of the same *application*. The *datasets* may contain a **description** so users can easily understand what is the content of the *dataset*. Also every *dataset* needs to contain **attributes** elaborated in more detail in the following section 2.3. Moreover, the *System users dataset* also needs to contain the *attributes* for the username and the password, as mentioned at the beginning of this section.

The following figure 2.2 shows the relation between the *datasets* and the database tables to which they were likened in section 1.2.

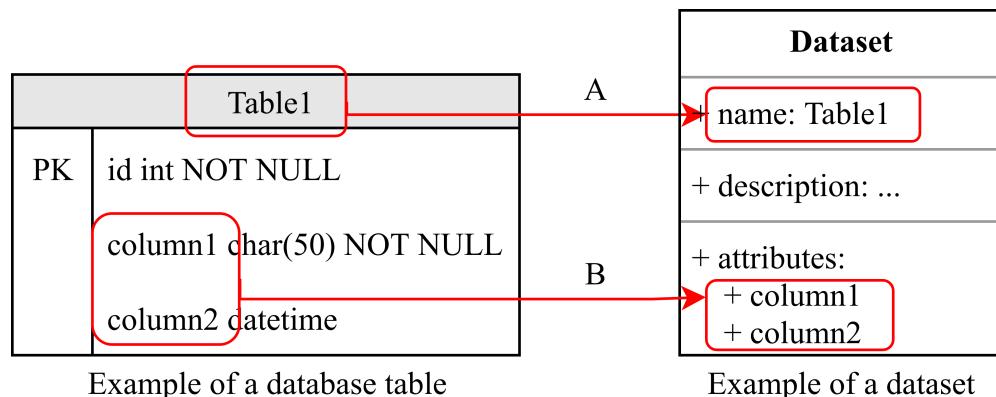


Figure 2.2: Relation between *datasets* and database tables.

As we can see from the figure 2.2, the database tables and the *datasets* share some elements—for example, a *dataset* name, which corresponds to a database table name (as shows arrow A) and the *attributes*, which correspond to the database table columns (as shows arrow B). In addition to this, the *datasets* also contain description. In the example database table, both columns also have additional properties, such as *char(50)*, *NOT NULL* or *datetime*. These properties will be elaborated in the following section 2.3, especially in the subsection 2.3.1.

## 2.3 Attributes

Every *dataset* needs to have at least one *attribute* (defined in section 1.2). A *dataset* without any *attributes* makes no sense—it can be compared to a database table without any columns. In the previous section 2.2, we have decided to distinguish between the definition of the *System datasets* and the *User-defined datasets*. The *System users dataset* needs to contain two significant *attributes* in order to be able to authenticate the user—the username and the password (more on this topic can be found in the Login credentials subsubsection of section 3.4). These two attributes need to be present in every *application descriptor* and thus need to be distinguishable from the other attributes. Without the username and the password attributes users would not be able to log into the *application*. This results in 3 categories of *attributes*:

**Password attribute** *Attribute* containing *application* specific settings of passwords for *application* users, values such as minimal and maximal length and additional security requirements need to be describable here. This *attribute* is valid only in the *System users dataset* and since the values are *application* specific, this *attribute* is required to be present.

**Username attribute** *Attribute* containing *application* specific settings of usernames for *application* users, values such as minimal and maximal length need to be describable here. This *attribute* is valid only in the *System users dataset* and since the values are *application* specific, this *attribute* is required to be present. It is also required that the username is always filled, because every user of the *application* needs a username to log in with.

**Basic attributes** Basic *attributes* are up to the creator of the *application*, all basic *attributes* are treated the same way and do not have any specific values. Basic *attributes* are valid for both *User-defined* and *System users dataset*.

With the knowledge of the *attributes* categories, we have to decide on their structure within the *datasets*. Solution for *User-defined datasets* is obvious since these *datasets* cannot contain *Password* and *Username attributes*. Because of this each *User-defined dataset* will contain a list of *Basic attributes*. The situation is different though for the *System users dataset*, because the *Password attribute* and the *Username attribute* need to be present. As a solution for this we have following options:

1. The *Username* and the *Password attributes* will be in the same list as *Basic attributes*. This will be possible only if there are some identifiers for both the *Username* and the *Password attributes*. With this approach, we will be able to easily find these two *attributes* and perform some operations on them if necessary.
2. The *Username* and the *Password attributes* will be separated from the *Basic attributes*. With this approach, it would be easy to distinguish these special *attributes* from the *Basic* ones just by the structure of the *application descriptor*. It would also be simpler for us to decide, whether we want to send to the client application values stored in these special *attributes*.
3. The last option is the middle way between the options 1 and 2. With this approach the *Password attribute* would be separated from the *Basic attributes* but the *Username attribute* would be on the same level as the *Basic attributes*. This idea is based on the fact, that we have not found a good reason for sending the password to the client applications, but the username could be a desired piece of information for the client. With this approach, the *Password attribute* would not get mixed with the *Basic attributes*. The *Username attribute* would be on the other hand a part of the *Basic attributes*. We suppose it would be sent to the client with the other information about the user. This also means that the *Username attribute* would need to have a special identifier to distinguish it from the *Basic attributes*.

From the provided ideas we have decided to select the approach 3. As described, this approach provides us the possibility to have the *Username attribute* together with the *Basic attributes* and at the same time to have the *Password attribute* separated from them.

The following figure 2.3 shows the approach 3 in a graphical form.

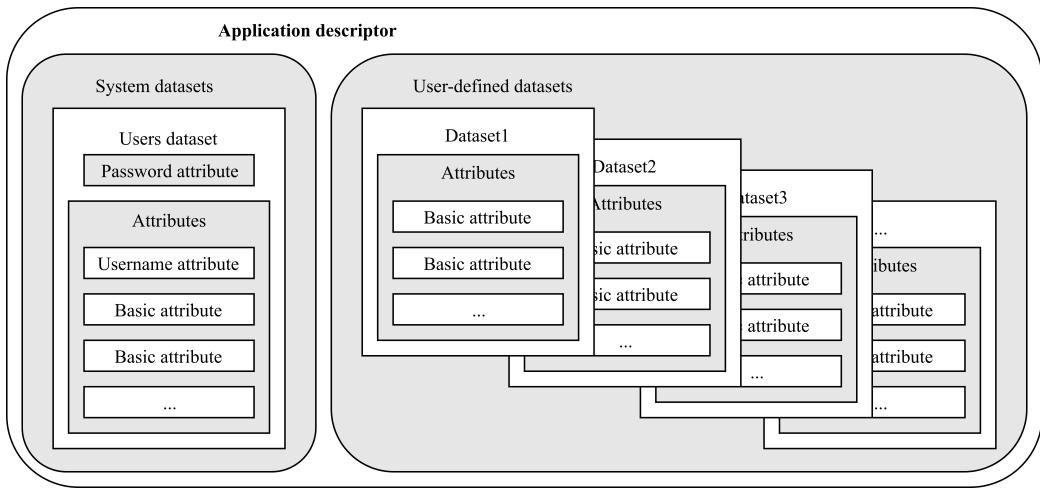


Figure 2.3: The application descriptor structure in a graphical form.

As we can see the *application descriptor* is divided into the *System datasets* and the *User-defined datasets* as explained in section 2.2. The *System datasets* contain the *Users dataset*, that implements the approach 3 to its *attributes* structure. On the other hand the *User-defined datasets* contain just the *Basic attributes*.

### 2.3.1 Complete list of available attribute properties

*Attribute properties*—defined in section 1.2—are used to define characteristics of an *attribute*. This list contains *properties* that we have found either crucial for the generated *application* to work or useful for the end users. Inspiration for these *properties* was taken from database columns and their basic data types as well as database constraints such as NOT NULL, FOREIGN KEY and UNIQUE. Also following the requirement R10 (1.3), we want the source code to be extendable, thus if this requirement is fulfilled, any missing *attribute properties* can be implemented later. For each *attribute property*, we will also demonstrate an example usage on one of the scenarios from the Representative scenarios subsubsection of section 1.1.

1. **Name** This string *property* will be required for every *attribute* and it is required that the name of the *attribute* is unique within the *dataset*. This allows us to distinguish each *attribute* within the *dataset*. For example, we can have in the *dataset* named *Books* an *attribute* with the name *property* set to *Name of the book*, as in the library application scenario from 2a.

2. **Description** This string *property* will be valid for every *attribute* and it will be optional. If filled it provides additional information about what should the value of the *attribute* contain to help the end users. For example it can describe the format of the string expected to be filled in. In the library application scenario from 2a a description *property* of the *attribute* named *Authors of the book* can be *Please fill 1–5 authors of the book*.
3. **Type** This *property* from enumeration of available types discussed later in this bullet will be required for each *attribute* and its value will depend on the category of the *attribute*. For example we can have a *dataset* named *My Dataset* with *attribute* named *My Attribute*. When creating a record into *My Dataset*, the type *property* of *My Attribute* restricts what values can be in *My Attribute* stored. More specific examples can be found for each type:
  - For the *Password attribute* (as defined in section 2.3) we have decided that the only valid value will be **password** since the *Password attribute* is a special type of *attribute*.
  - For the *Username attribute* (as defined in section 2.3) the only valid value will be **username**. This way we will be able to differentiate the *Username attribute* from the other *attributes*, since they will be in a single list as decided at the beginning of section 2.3 by approach 3.
  - *Basic attributes* can be likened to the user defined database columns (the relation to database column was mentioned in section 1.2 and the fact, that they are user defined is based on section 2.3). Database columns contain either data from a set of predefined types (numbers, strings, dates, etc.) or references to other database tables. Inspired by this parallel, the *Basic attributes* can also be either a *Basic type* or a *Reference type*:

**Basic type** The same way database columns must have data types, each *attribute* must have its type. This value determines the type of data the *attribute* accepts. An inspiration for the supported types was taken from the HTML5 input types (listed for example in article [HTM19]). Even though HTML5 offers a wide range of different types when creating *application* some types may not be implemented. Responsibility for solving this problem though depends on the final implementation, which is required to

have an easily extendable source code by the requirement R10 (1.3), thus provide an easy way to add support for the new types. The following enumeration contains types adopted from HTML5, that we have found useful to be supported. Each type also contains general examples of valid input values and a specific example related to the library representative scenario 2a.

**color** represents a hexadecimal color, for example `#ff00e6`. A specific example is a *dataset* with book statuses (present, borrowed, lost, etc.). Each of the statuses can have a color assigned, to visually distinguish between the book statuses. The *dataset* will thus contain an *attribute* called *Status color* with a type *attribute property* set to “color”.

**date** represents a date in the yyyy-MM-dd format, e.g. `2019-02-21`. For example, the *dataset* with books may contain an *attribute* with a date when the book was added to the library.

**datetime** represents a date and a time in the yyyy-MM-ddThh:mm format, e.g. `2019-02-11T20:57`. For example, the *dataset* with borrowings can contain an *attribute* with a date and a time of the borrowing.

**email** represents an email address in the x@y.z format, e.g. `example@email.com`. The *dataset* with readers could contain an *attribute* with an email address of the reader.

**month** represents a month in the yyyy-MM format, e.g. `2019-02`. For example, the *dataset* with payrolls can contain an *attribute* with the month the payroll belongs to.

**int** represents a signed or unsigned integer, e.g. `-1234`. The *dataset* with books can contain an *attribute* with an age restriction—number meaning the minimal age of the reader to be allowed to borrow the book.

**float** represents a signed or unsigned floating point number, e.g. `-12.34`. The *dataset* with books can contain an *attribute* with the original price of the book.

**year** represents any at most four-digit positive or negative number, e.g. `-42`. The *dataset* with books can contain an *attribute* with the year when the book was published.

**phone** represents any phone number with legal characters (numbers and symbols “+”, “(”, “)”, “.”, “-”, “ ” and

“,”), e.g. *+123 (456)-789*. In HTML5 this type is named `tel`, but we have found the name `phone` more suitable. The *dataset* with readers could contain an *attribute* with a phone number of the reader.

**string** represents any preferably short string, e.g. *Hello world!*. The *dataset* with books will probably contain an *attribute* with the name of the book, that will have its type set to “string”.

**time** represents a time in the hh:mm format, e.g. *14:19*. The *dataset* with library employees could contain an *attribute* with the start or end time of the employee’s shift.

**url** represents an absolute or relative url, e.g. *www.example.com*. The *dataset* with books can contain an *attribute* with a link to a store to buy the book.

**bool** represents a boolean value, e.g. *0* or *1*. Instead of 0 and 1, this value can be seen as true or false. For example, the *dataset* with readers can contain an *attribute* with the information whether the reader has paid the library fee.

**text** represents any preferably longer (multiline) string, e.g. *Hello,\r\nI am multiline text.\r\nLorem Ipsum ....* This type can be useful for example in the *dataset* with books in an *attribute* with the book description.

**Reference type** References can be likened to the FOREIGN KEY constraints in a database—value for the *attribute* is taken from any *dataset* in the same *application*. The crucial requirement for the reference is to point to a *dataset* in the same *application* in order to be valid, because only data within the same *application* are accessible to the end user. Because the name of each *dataset* is unique (this requirement was explained in section 2.2) this name will be used as the value of the type *property*.



Figure 2.4: Example of references between *dataset* A (authors) and *dataset* B (books).

We will illustrate an example on the library representative scenario from 2a shown in the figure 2.4. We have a *dataset* with books—named B—and a *dataset* with book authors—named A. Each book can have one or more authors and two books can have the same author. Because of this the *dataset* B will contain an *attribute* named BA, which will contain a list of authors of the book. This *attribute* will be of type *Reference type attribute* and will point to the *dataset* A, which means its type will be “A”—as the unique name of the *dataset* the reference is pointing to.

References can be also used to create enumerations. For this we need a *dataset* containing all valid values of the enumeration. This *dataset* can then be referenced in other *datasets* in *attributes* that should contain the value from the enumeration.

4. **Required** This boolean *property* will be valid for every *attribute* and states whether the *attribute* needs to have a value filled when creating or updating a record of a *dataset* that the *attribute* belongs to. The name nullable was also considered for this *property*, but was rejected at the end, because the meaning of the word “required” seemed more clear even for people without the knowledge of the meaning of the word NULL in the database environment. Specifically, the *Username* and the *Password* *attributes* must have the required *property* set to true, since no *application* user could be created without the username and the password. For the *Basic attributes*, this *property* is up to the creator of the *application*. For example, a *dataset* with books from a library representative scenario described in 2a will probably have an *attribute* name of the book with required *property* set to true since every book needs to have a name. On the other hand, the *attribute* with age restriction does not need to be filled for example for fairytales, thus this *attribute* would have the required *property* set to false.
5. **Unique** This boolean *attribute* will be valid only for the *Username attribute* and will be required to be set to true. It ensures that usernames within each *application* are unique. In the future, support for this *property* can be extended also to the *Basic attributes*, but this extention will not be part of this thesis.
6. **Min/Max** Validity and meaning of these integer *properties* will depend on the value of the type *property* of the *attribute*, already defined in this enumeration in the bullet 3. Meaning for each of the valid

type *property* values is explained in the following enumeration with an example usage for the ATS (applicant tracking system) representative scenario described in 1a.

- Basic types defining text values—text, string, username, password. For *attributes* with the type *property* set to one of these values the meaning is a minimal/maximal length of the input string. If the value of the min *property* is set to 5, then the valid inputs are for example “abcde” or “abcdefg”, but not “ab” or “abcd”, because they are too short. The min *property* is useful for example for setting the minimal required length of a password. This can be done by setting the min *property* to the *Password attribute*. If a value of the max *property* is set to 5, then valid inputs are for example “a” or “abcde”, but not “abcdef” or “abcdefg”, because they are too long. This *property* is useful for example for setting the maximal valid length of the description of candidate hobbies and interests.
- Basic types defining numeric values—int, float, year. For *attributes* with the type *property* set to one of these values, the meaning is minimum/maximum value of the number. If the value of the min *property* is set to -3, then the valid input is -3 or any greater value. If the value of the max *property* is set to 5, then the valid input is 5 or any smaller value. For example, if we store a candidate, we can require their age to be at least 15 years or if we store a salary with a job, we can set its maximum value. Or if we want to store the latitude and the longitude, we can restrict the values to -90–90 and -180–180 respectively.
- For any Reference type *attribute* the number set to the min *property* means the minimum number of references that must be selected in the *attribute* value. If the value of the min *property* is set to 5, then a valid number of references is 5, 6, or more and invalid is 4, 3, and less. The number set to the max *property* means the maximum of references that can be selected in the *attribute* value. If the value of the max *property* is set to 5, then a valid number of references is 5, 4, or less and invalid is 6, 7, and more. For example, we can have a *dataset* containing specializations, such as engineering, IT, pharmacy, legal, etc. Each candidate can have several specializations assigned—these values will be references to the *dataset* with specializations and we can limit, that each candidate must have at least 2 and at most 5 specializations.

- Other Basic types—color, date, datetime, email, month, phone, time, url, bool). For these types the min/max *property* is not supported. The reason is, that these values require different validations and it is not clear what the minimum/maximum value should mean.

The other approach contained *properties* minNumber/maxNumber, minChar/maxChar and minReference/maxReference valid based on *attribute* type *property*, but since each *attribute* can have only one type *property*, we have decided to merge them into the min/max *properties* respectively.

7. **OnDeleteAction** This *property* with valid values enumerated later in this bullet is required for every Reference type *attribute* (defined in the bullet 3) and it will indicate what should happen if a record referenced in this *attribute* is deleted. For the Basic type *attributes* (defined in the bullet 3) the *property* need not to be set and if it is set, only the *none* value is valid. We will introduce a simple example, on which we will demonstrate different action results in the following enumeration.

The example is illustrated in figure 2.5, which was already shown when elaboration the Type *property* of the Reference type *attributes* in the bullet 3. In this example there is a *dataset* with book authors called A and a *dataset* with books called B. The *dataset* A has only one *attribute* containing a name of the author with the name *property* set to AN and the type *property* set to string. The *dataset* B contains two *attributes*—*attribute* with a name of the book, with the name *property* set to BN and the type *property* set to string and an *attribute* with authors of the book with the name *property* set to BA and the type *property* set to A—meaning a reference to the *dataset* A as defined in the bullet 3. The other *properties* as well as other *attributes* and *datasets* are not important in this example. There are two records in the *dataset* A, as well as in the *dataset* B and each of them has a unique identifier used when the value is referenced and we will also use this identifier when talking about the record..

B (books)			A (authors)	
identifier	BN	BA	identifier	AN
20	B1	10,11	10	A1
21	B2	10		A2

Figure 2.5: Example containing *dataset* A (authors) and *dataset* B (books).

As we can see in *dataset B*, a record 20 references records 10 and 11 from the *dataset A* and a record 21 references a record 10 from the *dataset A*. If we want to delete a record from *dataset B*, there is no problem, since no record from the *dataset B* is referenced anywhere else. On the other hand, if we want to delete a record from *dataset A*, we encounter a problem—record 10 is referenced twice in *dataset B* and record 11 once. To solve this problem, we have decided to implement the *onDeleteAction* *property* with following available actions:

**none** Represents undefined *onDeleteAction*. This value is for every *attribute* of Basic type, because for Basic type *attributes*, transitive deletion makes no sense. On the other hand this value is forbidden for the *attributes* of Reference type, because the deletion action must be defined.

**cascade** When a record referenced in *attribute* with *onDeleteAction* set to cascade is deleted, the record having it as *attribute* value will be deleted as well. Cascade cannot be used for *Basic type attributes*. Also, it cannot be used in *attributes* of type *System users dataset* reference to ensure that the last user of the *application* cannot be accidentally deleted.

From the example, when the *attribute* BA in the *dataset B* has *onDeleteAction* set to cascade, if we delete the record 10 in the *dataset A*, both records 20 and 21 in the *dataset B* will be deleted, because the record 10 was referenced in both of them. The result after deleting the record 10 can be seen in the figure 2.6. If we delete the record 11 in the *dataset A*, then only the record 20 in the *dataset B* will be deleted since the record 21 is not referencing the record 11. The result after deleting the record 11 is displayed in the figure 2.7.

B (books)			A (authors)	
identifier	BN	BA	identifier	AN
			11	A2

Figure 2.6: Result after deleting the record with identifier 10 when BA has the *onDeleteAction* set to cascade.

B (books)			A (authors)	
identifier	BN	BA	identifier	AN
21	B2	10	→ 10	A1

Figure 2.7: Result after deleting the record with identifier 11 when BA has the onDeleteAction set to cascade.

An example of the cascade onDeleteAction usage is in the ATS (applicant tracking system) scenario from 1a. If a candidate in the European Union wants to be deleted from the database, it is important to remove all records linked to them as well, by the rules of GDPR (meaning General Data Protection Regulation, with more detail on the official site [Eug]).

**setEmpty** When a record referenced in the *attribute* with onDeleteAction set to setEmpty is deleted, the to-be-deleted reference value in the record will be replaced with an empty value—in other words removed. The only exception is in a situation when setting an empty value would break the minimal amount required by the min *property* (defined in the bullet 6) or if it would remove the last value when the required *property* is set to true (defined in the bullet 4). In both of these cases no record can be deleted. SetEmpty cannot be used for the *Basic type attributes*.

From the example, if we want to delete a record 10 when the onDeleteAction of the *attribute* BA is set to setEmpty, we need to consider the values set to min and required *properties* as mentioned in the previous paragraph. If the min *property* is set to 1 or the required *property* is set to true, then no deletion can be performed, since deleting the record 10 would remove the last reference in the *attribute* BA for the record 21 and thus violate the min and/or required *property*. If the min and required *properties* are not set, the record 10 can be deleted and its references removed from the records 20 and 21. The result after the deletion of the record 10 is in the figure 2.8. Since the record 21 contains just 1 reference, the min *property* of the *attribute* BA cannot be greater than 1, thus settings for the min or required *properties* does not matter when deleting the record 11—in all the cases both records 20 and 21 will still contain a reference to the record 10. Deleting the record 11 when onDeleteAction of *attribute* BA

is set to `setEmpty` will result in the deletion of the record 11 and a removal of its reference from the record 20. Result of this action is in the figure 2.9.

B (books)			A (authors)	
identifier	BN	BA	identifier	AN
20	B1	11	11	A2
21	B2			

Figure 2.8: Result after deleting the record with an identifier 10 when BA has the `onDeleteAction` set to `setEmpty` and both `min` and required *properties* are not set.

B (books)			A (authors)	
identifier	BN	BA	identifier	AN
20	B1	10	10	A1
21	B2	10		

Figure 2.9: Result after deleting the record with an identifier 10 when BA has the `onDeleteAction` set to `setEmpty`.

An example of `setEmpty` `onDeleteAction` is in the ATS (applicant tracking system) example from the section 1a. If we have a table with interviews and jobs and a job gets deleted, we want the job to be removed from the interview, but we do not want to remove the whole interview, because it may contain other important information.

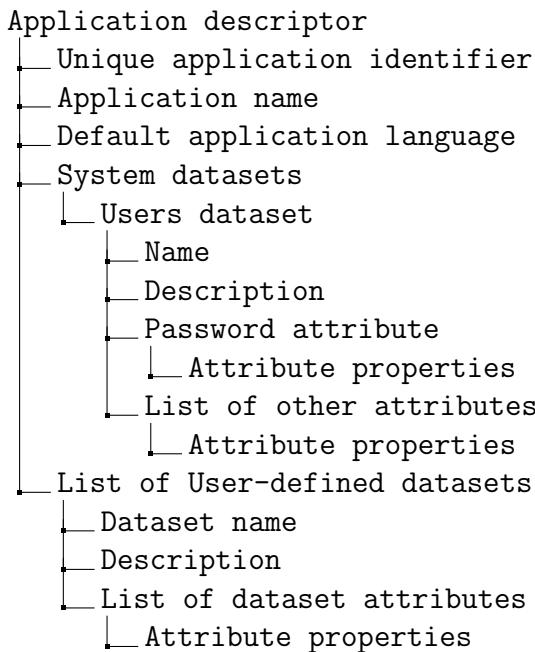
**protect** When there is an attempt to delete record referenced in an *attribute* with `onDeleteAction` set to `protect`, the deletion is stopped due to this protection and none of the records is deleted. Protect cannot be used for *Basic type attributes*. The result after the attempt to delete the record 10 or 11 with the `onDeleteAction` of the *attribute* BA set to `protect` is the same as before the attempt. This means, that the figure 2.5 does not change.

An example of the `protect` `onDeleteAction` is in the library example in 2a. If we have a dataset with authors and books and we delete an author, we do not necessarily want to delete all the books as well. Instead, we should prevent the deletion.

8. **Safer** This optional boolean *property* will be valid only for the *Password attribute*. If set to true, additional requirements for password safety are enforced. If this *property* is not set for the *Password attribute*, false is automatically set and no additional safety for the password is required. The other approach would be to have a minNumber, maxNumber, minLowerCase, maxLowerCase, minUpperCase, and maxUpperCase *properties*, but at the end, we have decided to merge them all into the safer *property* for easier creation and clearness of the *application descriptor*. More information about password security can be found in the Password policy subsubsection of section 3.4.

## 2.4 Application descriptor structure

As a result of analysis from the sections 2.1, 2.2 and 2.3, we have a basic structure of the *application descriptor*. This structure contains all the necessary information that we need to generate an *application*.



## 2.5 Application descriptor validation

It is important to ensure, that only valid *application descriptors* are accepted by the *application generating software*. Because of that we need to be able to validate the *application descriptor*. This section contains list of validations,

that need to be performed, before the *application descriptor* can be marked as correct.

### 2.5.1 Necessary validations

- *Application descriptor* structure – required fields and their types, no other than available fields (as defined in section 2.4)
- At least one *User-defined dataset* (as explained in section 2.2)
- A uniqueness of the application identifier (as explained in section 2.1)
- A uniqueness of the *dataset* names within the application (as explained in section 2.2)
- The names of the *datasets* cannot be the same as names of the Basic types (defined in the bullet 3 of subsection 2.3.1). Without this constraint we would not be able to correctly determine whether the *attribute* is of the Basic type or of the Reference type to a *dataset* named as the Basic type.
- At least one required *attribute* in each *dataset*. As stated in section 2.3 we need at least one *attribute* in each *dataset*. On top of that we need the *attribute* to have the required *property* set to true, thus users of the *application* would not be able to create a completely empty record in a *dataset*.
- The *attributes* within the *dataset* must have unique values in the name *properties*—meaning unique names (as explained in the bullet 1 of subsection 2.3.1)
- The *application descriptor* cannot contain invalid references to the *datasets* that do not exist (as explained in the bullet 3 of subsection 2.3.1)
- Reference type *attributes* must have the *onDeleteAction* *property* specified (as explained in the bullet 7 of subsection 2.3.1). If the reference is to a *System users dataset*, then the *onDeleteAction* *property* cannot be set to cascade (as explained in the bullet 7 of subsection 2.3.1)
- *Password attribute* must have its type *property* set to “password” and the required *property* to true (as explained in the bullet 3 of subsection 2.3.1)

- No other than *Password attribute* can have its type *property* set to “password” (as explained in the bullet 3 of subsection 2.3.1)
- No *attribute* in the *User-defined datasets* can have the type *property* set to “username” (as explained in the bullet 3 of subsection 2.3.1)
- There must be exactly one *attribute* with the type *property* set to “username” in the *System users dataset* (as explained in the bullet 3 of subsection 2.3.1)
- The *Username attribute* must have the required *property* set to true (as explained in section 2.3) and the unique *property* set to true (as explained in the bullet 5 of subsection 2.3.1)
- The *Basic type attributes* cannot have the onDeleteAction *property* value set to anything but none (as explained in the bullet 7 of subsection 2.3.1)
- The safer *property* can be set only for the *Password attribute* (as explained in the bullet 8 of subsection 2.3.1)
- The min and max *properties* cannot be set for the *attributes* with the type *property* set to “color”, “date”, “datetime”, “email”, “month”, “phone”, “time”, “url” or “bool” (as explained in the bullet 6 of subsection 2.3.1)
- Value of the min *property* must be less than or equal to the value of the max *property* if both are set (this results from the logical meaning of the *properties* defined in the bullet 6 of subsection 2.3.1)
- Values of the min and max *properties* must be greater than 0 for attributes with the type *property* set to “text”, “string”, “username”, “password” and for the Reference type *attributes*, because length of a string and a number of references cannot be a negative number (the meaning of min and max *properties* was defined in the bullet 6 of subsection 2.3.1)
- If the min *property* is set, the required *property* cannot be set to false for the Reference type *attributes* since having a required minimum number of records implies that the *attribute* must be filled. Setting the required *property* to true implies that the *attribute* must have at least 1 reference filled and thus set the min *property* to at least 1.

## 2.6 Summary and next steps

In this chapter, we have designed the structure and the necessary parts of the *application descriptor*. This design will be used to select a suitable format of the *application descriptor*. Then we use it to describe an *application* and generate it with the *application generating software*.

# 3. Implementation analysis

Before analyzing various approaches of how to implement a software described in section 1.5, it would be a good idea to assign this software a name. Since this software will be creating a meta (with the meaning about the thing itself, in this case, a software about applications) applications, first part of the name will be Meta and since the company mentioned in the Original idea subsubsection in chapter 1, that has reached us wanted a request management system, the second part of the name will be an abbreviation of it—RMS. This together gives the name—MetaRMS.

This chapter analyses various approaches to MetaRMS implementation. In each subsection, we will consider solutions for individual problems, that need to be solved before the software implementation.

## 3.1 Software architecture

As the first implementation analysis point, we had to design and decide about the software architecture. As stated in section 1.3 MetaRMS is not required to work offline. Also, it is expected, that multiple users will connect to the same application at the same moment from different devices. Because of this, we need to have a central endpoint with data storage to which the users will connect. This results in a client-server architectural pattern, which is well described in article [Cli19]—this pattern consists of two parties, one server (our central endpoint) and multiple clients, communication over a computer network.

The following figure 3.1 shows the key parts of client-server architecture. Description of each part with more details will follow the figure.

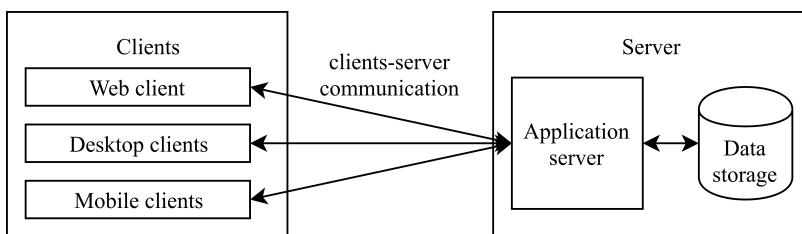


Figure 3.1: Basic client-server architecture suitable for MetaRMS.

**Server** We have decided to divide server part to the following subparts:

**Data storage** This storage will be implemented as a relational database. It needs to be able to store login credentials for

application users, information about generated applications and their data. This data storage will be elaborated in more detail in section 3.8.

**Application server** This component will be an application running on a server accessible on the Internet so the client applications can connect to it. It will provide authentication for users logging from any of the client applications and will also be responsible for the input data validations and authorization checks.

The reason for separating the data storage and the application server is based on requirement R10 (1.3), in which we want MetaRMS to have an easily extendable source code. Thus if we want to switch to another database system, allowing us, for example, additional options for data handling, it will be easier to have the database separated from the application.

**Clients** We expect, that MetaRMS users will connect to the server from various devices because of the multiplatform support, required in requirement R2 (1.3). Due to that, we have divided client applications into the following groups:

**Web client** This client will be accessible from both desktop and mobile devices via a web browser and will provide a user-friendly way to create a new application, log into it and perform CRUD operations on the data.

**Desktop clients** This client applications will run on PCs and laptops with different operating systems. Requirement R9 (1.3) ensures, that these applications can be created and can connect to the MetaRMS server.

**Mobile clients** This client applications will run on mobile devices with different operating systems. Requirement R9 (1.3) ensures, that these applications can be created and can connect to the MetaRMS server.

Since our resources are limited, we have decided, that only the implementation of the Web client application will be a part of this thesis. This client will test the connection to MetaRMS and will be the main access point to it. The Web client was selected because it allows connecting from different devices, independent of their type and operating system. Because of requirement R9 (1.3), additional client applications mentioned in the enumeration above can be created later.

**Communication between clients and server** This is another important part of the architecture. At first, we need to define a coupling between the clients and the server. Coupling is closely related to the selection of an API style, which is one of our requirements—requirement R9 (1.3). As a great source of information about API styles has proven the following article [Nal18].

**Tightly coupled** The term tight coupling generally means, that one software is dependant on the other one extensively. If the communication happens via the Internet, it can be realized for example by a Remote Procedure Call (RPC) over HTTP(S). The tightly coupled client-server communication would be suitable if we develop and maintain the server and all the client applications, but this is in conflict with requirement R9 (1.3) since we want to allow other developers to use it as an endpoint for their own client applications. Also in tightly coupled systems, it is more difficult to change or update the code, which may cause problems to requirement R10 (1.3).

**Loosely coupled** The term loosely coupled is the opposite of the previously mentioned tightly coupled. In this case, one software is on the other one dependant as least as possible. This approach allows easier code modifications for the future bug fixes and adding new features. This is what is required by requirement R10 (1.3). Also when the client applications are not dependant on the server and the server interface is publicly accessible, anyone is able to create an own, platform independent, client application, just like in requirement R9 (1.3) and supporting requirement R2 (1.3).

From the approaches to the client-server communication we have opted for a loosely coupled communication which supports requirement R10 (1.3). Since we want to provide a public web API as in requirement R9 (1.3), based on recommendations in the article [Nal18] we have decided to implement it as HTTP(S)/JSON API. JSON as the data format is platform independent, thus supports requirement R2 (1.3). We would also like to support communication over HTTPS to protect it against attackers as mentioned in the article [Kay19].

In the final solution, we have decided to merge the Web client and the Application server into one project running always simultaneously. This was because since no other client application yet exists, running only the Application server part makes no sense. Also thanks to this merge, we have

to deploy the project only once and also only one server is required for it to run. Even though both of the applications run on the same machine, the Web client still connects to the Application server API endpoints to keep it as an illustration of how to implement such a connection in other clients. Figure 3.2 illustrates the final state of the MetaRMS architecture.

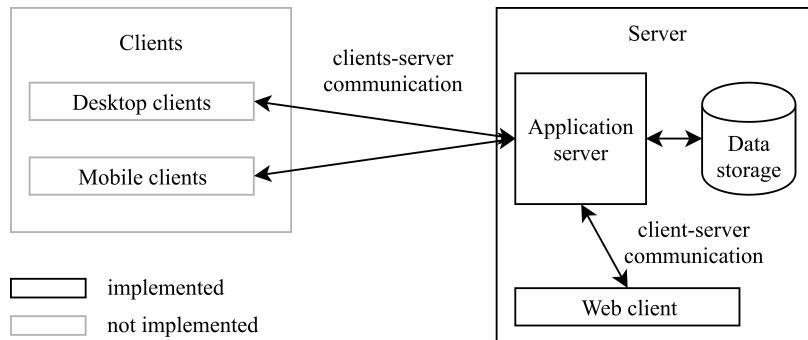


Figure 3.2: Final MetaRMS architecture.

## 3.2 Language and environment

With software layout in mind, it is time to pick the right language for developing a web application with most of the code running on the server. Since we want easy code extensibility as stated in requirement R10 (1.3), we are looking for a widely used and popular language with a great community. Not many of server-side programming languages have a perceptible market share as seen from W3Tech's technology overview [W3r] and thus we are picking from:

1. PHP as a scripting language with a possibility of using one of the PHP frameworks – this is the most popular option between developers, it is free for use, has a large community, many tutorials, is cross-platform and open source.
2. ASP.NET framework with a possibility of using any .NET language. Unfortunately, only supported server operating system is Windows, which breaks requirement R2 (1.3). While searching for information about ASP.NET framework, we have discovered ASP.NET Core framework, which will be elaborated in the next bullet.
3. ASP.NET Core framework with a possibility of using any .NET language. Although this framework was not present in the list of the most frequently used server-side languages/frameworks it is probably

included in the ASP.NET framework in general since they have a lot of things in common. We will though focus on their differences, the main one is, that ASP.NET Core needs just .NET Core framework and because of this is ASP.NET Core multiplatform. In addition to this, it is also open source and free, thus accessible to anyone.

The possibility of using C# as a programming language was crucial when picking ASP.NET Core over PHP because the author of the thesis has more experience with it. Moreover, with the existence of ASP.NET Core and its ability to run multiplatform, the requirement R2 (1.3) for MetaRMS to be multiplatform will not be violated as well as the possibility of using Unix type system for developing the software. With the knowledge of the framework and the programming language, we can go through the software architecture once again and determine some platform-specific details.

Application server described in section 3.1 can be divided into several layers. Each of these layers is responsible for specific tasks and thus can be implemented differently. The following enumeration lists technologies we will use for these different layers.

**Service layer** This layer is responsible for providing the JSON web API as selected in section 3.1 when elaborating coupleness of the server and the clients. This layer will be written in C# on the ASP.NET Core platform and will contain controllers accepting requests from the client applications.

**Business logic layer** This layer is responsible for the business logic on the server and will be as well written in C# on the ASP.NET Core platform. Business logic consists of function for the authorization of users, data validations, etc.

**Data access layer** This layer is responsible for the communication between the application server and the database. Selection of this layer is based on the selection of the database and will be in detail elaborated in section 3.8.

As decided in section 3.1 we will implement the Web client application. As the name suggests, this client application will be accessible through a web browser. With the use of the JSON web API, we are not limited to a specific framework. Since ASP.NET Core provides a page-focused framework for building dynamic web sites called Razor pages, we have decided to choose it as our front-end framework. Also the Microsoft website [Ric19] provides tutorials on how to implement the Razor pages front-end.

The Web client application will consist of the following layers:

**Presentation layer** The function of this layer is to display the application data in the web browser to the end users. Data to this layer will be provided through the JSON web API calls to the MetaRMS server. As selected at the beginning of this enumeration this layer will be implemented in Razor pages framework with the help of the front-end libraries Bootstrap and Bootstrap-select. This presentation layer should be flexible enough to satisfy both desktop and mobile users.

**Business logic layer** This layer can contain some parts of the business logic such as validations. These validations can be therefore performed on the client side before the data are sent to the server.

Since we want this web client to be displayed well on both desktop and mobile devices, we have decided to take advantage of the front-end library Bootstrap ([getbootstrap.com](http://getbootstrap.com)) with the bootstrap-select plugin ([developer.snapappointments.com/bootstrap-select/](http://developer.snapappointments.com/bootstrap-select/)). Bootstrap is currently one of the most popular front-end component libraries. It is open source, easy to use and well documented with a lot of tutorials available online. Bootstrap-select is a plugin that enhances select elements with multiselection and searching. Bootstrap with the bootstrap-select plugin was selected not only because of its popularity and a wide selection of tutorials but also because the author of the thesis has experience with using them.

### 3.3 References

When creating an application descriptor, each attribute needs to have its type set, as stated in the bullet 3 of subsection 2.3.1. If the attribute is of *Basic type*, it is clear that when storing a record of that dataset, we will store the actual value of the attribute. This is not so obvious for *Reference type* attributes. We need to always remember where the reference is pointing so we need to know the id of the referenced record.

When displaying a record with reference we also need to know the value of the reference, because users will not be interested in seeing an id, they will expect the real value. To save additional requests to the server we have decided to return deep copies of the data—an id accompanied with the value to be displayed. Without this, it would be necessary to make another request to the server for every reference received to load the value of the reference so that the client application can display it.

For storing references we thought of two approaches:

1. Store both id and its value with the record containing the reference. This approach makes it easy to get the value of the reference, thus

the get operation will be performed quickly and we will not need to trace the referenced value by its id. On the other hand, if the value of the referenced record is changed, we will need to find all places where the value is referenced and repair the value. Because of it, the patch operation will take a longer time.

2. Store only id of the reference in the record. By storing just the id we save not only time to update the referenced values but also space in the database, because the value will not be duplicated. The only disadvantage of this approach is the need to trace the referenced id to get its value.

From the above-mentioned approaches, we have decided to implement approach 2 because tracing an id only when users ask for it will be less demanding on the database.

### Displayed references depth

With references, a problem emerges. We have to decide how will the displayed value of one record from any dataset look. From section 2.3 we know that a dataset can have an arbitrary number of attributes. Because we expect that the important information will be between the first attributes we have decided to display values of the first at most 3 attributes. From our point of view, the first 3 attributes should be enough to distinguish between the records of one dataset.

In case when one of the first 3 attributes is of Reference type, we will build the value of the reference the same way—by combining values of the first 3 attributes of the referenced value. By applying this approach recursively, we could build an extremely long text representation of the reference. To prevent that we have decided that the maximal depth of this recursion will be 3. We have found this depth to be enough and show it on the ATS (applicant tracking system) 1a from our representative scenarios.

The ATS will probably contain a table with candidates, jobs, interviews, and specializations. Candidates will have a string name, string surname, and specialization referenced from the table with specializations. Jobs will have a string name and specialization referenced from the table with specializations. Interviews will have candidate referenced from the table with candidates and job referenced from the table with jobs. When displaying a record from the table with interviews we find the name and surname of the candidate and name of the job as the most important information. All of the values can be loaded from the references and since they are just plain strings, it will be displayed. Both the specialization of the candidate and job are

additional information that are not crucial when distinguishing between different candidates or jobs and this information can be seen at the detail of the record. Still, this value will be displayed at the reference value. From this, we can see that the deeper we go in the references, the less important the information is.

We have came to the same conclusion for other representative scenarios from the Representative scenarios subsubsection of section 1.1 as well.

## 3.4 Authentication

By our requirement R6 (1.3) the applications generated by MetaRMS must contain authentication to ensure that only users registered in the application can access the data. In the following subsubsections, we will elaborate on some details important for the authentication implementation.

### Login credentials

MetaRMS will host many different applications into which users can log. During the login process, it is important to identify the user attempting to log in to be able to check if the inserted password is correct. Because of this, we have found two approaches on how to identify the user:

1. Using username and password only. This approach will cause that there could not be two users with the same username even if they are in different applications. Especially it would, for example, mean, that each generated application would need to have an administrator with a different username.
2. Using username, password and unique application identifier, from application descriptor (defined as required in section 2.3 and in section 2.1). Using both username and the application identifier would mean that usernames must be unique only within one generated application and that users in later generated applications would have the same amount of usernames to choose from as users in former generated applications.

From these two options, we have chosen using the application identifier together with the username as the unique user identifier. The option 2 was selected because of the possibility to have more users with the same username if they are in different applications since users of later created applications would otherwise have a limited selection of available usernames.

## Authentication method

With login credentials selected we have to decide about the process of sending them to the server. Without this, the user cannot be authenticated and thus we cannot identify them. Examples of these methods are mentioned in the article [Ger15].

1. HTTP Basic authentication is the simplest possible approach to enforce authentication of users. In this method the login credentials are sent with every request in `Authorization` header with the `Basic` keyword and are base64-encoded. The main disadvantages are that the login credentials might get exposed by sending them with every request, the credentials cannot expire and the password would need to be stored in the client application, which is another security issue.
2. In Cookies/Session-based authentication the server creates a session after the user logs in. The session id is sent to the client, which stores it in a cookie and sends it back to the server with every request. Thanks to the session id the server can find the state of the logged user. The disadvantage is, that cookies might be vulnerable to the CSRF (Cross-site Request Forgery) attacks, described in detail in article [Wika].
3. In Token-based authentication the server creates a signed token and sends it to the client. The client stores the token and sends it to the server with every request. Since the token is signed, it can be validated on the server side. The token can also contain additional information—claims. These claims can specify the expiration time or can identify the user. The biggest difference over the Cookies/Session-based authentication is, that the server does not have to store any information, thus is stateless. The disadvantage is, that tokens stored in the Web Storage might be vulnerable to the XSS (Cross-site Scripting), described in detail in article [Wikb].
4. One-Time Passwords is an approach which relies on another medium to authenticate the user. After submitting the login credentials, the user receives a code to an email address or a mobile phone. This code is then submitted to the client application and verified on the server. In this approach, the user's identity is ensured by the communication medium (email, mobile phone) since it is owned by the user. An example of this authentication approach is logging into a bank account, where you receive an SMS code.

When selecting the authentication method, we have been deciding between the Cookies/Session-based and the Token-based approaches. These

two approaches seemed secure enough while they did not require the implementation of additional services, such as sending SMS. We have found the article [Adn16] helpful for this decision. It mentions that the Token-based approach is more scalable, stateless, performant and mobile-ready, which is important for the future mobile client applications. Because of that, we have decided to implement approach 3. Also the article [Pro16] recommends using the HTTPS/SSL to prevent the man-in-the-middle attacks.

As recommended in the articles we have decided to implement the Token-based authentication in the form of a JWT (JSON web token). The format of the token was already described in the article and the detail description can be found on its homepage [jwt.io](https://jwt.io). The advantage of using JWT is its support by ASP.NET Core because a middleware for JWT exists in the `Microsoft.AspNetCore.Authentication.JwtBearer` package.

The final selected authentication method thus is the JWT token. After receiving valid login credentials the MetaRMS server application will send this JWT token to the client application. Then with each request from the client application the token will be added to the `Authorization` header of the request with the `Bearer` keyword.

## Password policy

In this subsection, we will make use of the safer property defined in the bullet 8 of subsection 2.3.1. It was already mentioned, that this optional property for the Password attribute will contain boolean value and if set to true, additional requirements for the password security will be enforced. As the additional properties, we can choose from a minimal length of the password or enforcing some characters to be present in each password. We have decided, that the default minimal length of the password with the safer property set to true will be 8 characters. This value can be overwritten but only to a higher value by the min property defined in the bullet 6 of subsection 2.3.1. Because of that the password with the safer property set to true must be 8 or more characters long. As regards enforcing some characters, we have decided to require at least 1 lower-case and 1 upper-case letter and either 1 number or 1 special non-word character to be present. Setting these policies makes the passwords harder to guess by the potential attacker. These properties were selected based on the article [Wil16] as well as their values.

## Password storing

As mentioned in section 3.1 a relational database data storage will be present on the server side of MetaRMS. In this database, we need to store user

credentials mentioned in the Login credentials subsubsection of section 3.4 to be able to authenticate the user. The username and the application identifier can be stored in a plain text, but it does not apply for the passwords. If an attacker gets access to the database with passwords stored as a plain text, they would get access to all the applications. Because of that, we have decided to store the passwords in a hashed form together with a random string—a salt. This way lookup tables and rainbow tables could not be used for the decryption of passwords as described in the article [Sal]. This random string is prepended to the password and together they are hashed. The final hash is then stored into the database and used for the user authentication. Using salt ensures that even if two users would have the same passwords, their hashes would be different.

## 3.5 Authorization

As in one of our safety requirements, the requirement R7 (1.3), it is necessary that the generated applications contain authorization. Thanks to the authorization users of the generated applications will be able to perform CRUD operations only on the data they are allowed to. Since MetaRMS is designed with the goal to be able to generate business applications we have decided to implement the authorization as role-based. Because of this individual users will be assigned with a set of named rights—a role—which will be used to authorize the user. This provides the administrator with an option to divide users into groups authorized to the same tasks (for example multiple recruiters in example 1a or multiple librarians in example 2a). On the other hand, this approach does not limit administrators from assigning each user different rights by assigning them a different set of rights.

### Granularity

Since we have decided, that users will be assigned with a set of rights we have to set the granularity of the rights and by the granularity we mean what will be the smallest unit, that can have the rights assigned. With the knowledge of application structure from section 2.4 there are several options:

1. Rights for the whole dataset. This way administrators will not be able to assign rights for individual attributes but only to the whole datasets. On the other hand, the creation and editing of rights will be much easier.
2. Rights for individual dataset attributes. This approach will provide a greater granularity to express the rights.

From the options we have decided to implement approach 1. The main reason was, that creating a set of rights with rights for each individual attribute in each dataset would be a tedious work for the administrator. Also, we expect, that values of attributes in the dataset are somehow related, thus the access should be granted to the dataset as a whole. For example, a librarian from our representative scenario 2a should have access to a dataset with books and should be able to read all information about the book as well as edit all the information in case of changes. The same applies to a recruiter in the ATS (applicant tracking system) representative scenario 1a—they should be able to fill all information about a new candidate as well as edit it all.

## Representation

For each dataset, we have to create a representation of the rights. This representation must express all of the CRUD operations and state whether the operation is on the dataset allowed or not. For this we came up with the following approaches:

1. Represent each of the CRUD operations individually. By this, we mean that for each dataset the rights will contain information whether each of the CRUD operations is allowed or not. The approach can be of course simplified and contain only allowed operations. This might get difficult to validate because for example allowing delete operation and forbidding read operation on the same dataset does not make sense.
2. Create a hierarchy of CRUD operations. With a logical hierarchy of CRUD operations, we will be able to assign each operation minimum required value that the user's rights need to have to allow them to perform the operation. This approach will simplify the authorization process and will also require fewer data to be stored within the rights.

We have decided to implement approach 2 because it is easier to check that the rights are valid, which results in easier usage when creating and modifying the rights. This way only one value needs to be associated with each dataset. This approach requires that we define the hierarchy of the CRUD operations. We have chosen the following order:

1. None – User with None rights to a dataset is not allowed to access the dataset at all. This means none of the CRUD operations can be performed on that dataset.
2. R – Dataset with rights set to R can be only read. The Create, Update and Delete operations are forbidden for a user with this rights.

Selecting Read as the first allowed operation is obvious since without read users will not be able to see the data.

3. CR – Dataset with rights set to CR allows creating new records and reading existing ones. The Update and Delete operations are forbidden for a user with this rights. Create as the second allowed operation was selected because updating an already existing record seemed as more invasive operation since it can damage already existing data.
4. CRU – Records in a dataset with rights set to CRU can be read, updated and created. The Delete operation is forbidden for a user with this rights. Update was selected as the third allowed operation since it is less destructive than the Delete operation, thus safer to use.
5. CRUD – When rights are set to CRUD, all of the CRUD operations can be performed with the data in the dataset. Delete is the last allowed operation since it allows complete removal of stored data, which we consider as the most dangerous operation.

### Rights validation

Even though we have opted for the approach 2 in the previous subsubsection there is still one validation we need to perform when the rights are created or modified. In the application descriptor, the attributes can be of a *Reference type*, pointing to another dataset within the same application as already mentioned in the bullet 3 of subsection 2.3.1. Imagine a situation with two datasets, *Dataset A* and *Dataset B*. Dataset A contains an attribute of type reference to Dataset B. If we set rights R, RU, CRU or CRUD to the Dataset A we also need to set at least rights R to the Dataset B within one rights set. Otherwise, we would read values from the Dataset B referenced in the Dataset A, which we do not have rights to read. This gives us the following necessary validation:

- Within one rights set for every dataset A with rights R, RU, CRU or CRUD, every dataset referenced in A needs to have at least R rights.

## 3.6 Messages handling

When end users are using the application, it is important to inform them whether their operations were successful or not. This makes the application more user-friendly as we need by the requirement R3 (1.3). To inform users about the results of their actions we need to think over the representation of

such messages. It is important to think about the crucial parts every message should contain and it is its type (such as an Info message, a Warning message or an Error message), text of the message and its code. Since these messages will be also used to inform the user if the input was invalid, we have decided to include an optional name of the attribute that the message regards to.

Since every application will have a different structure, the messages for each application will differ as well. Because of that, some parts of the messages will contain placeholders, that will be replaced by a value based on the application. These values are for example names of datasets or attributes or required length of the input. Because of that every message also needs a list of values, that will be replacing the placeholders. To specify the location of the placeholders we have decided to mark them by {number of a value from the list of placeholders}—number in curly braces.

For example, if we have a dataset A with attribute A1 of type int and the user inserts string, an error message would have its type set to Error, text to “Attribute {0} in dataset {1} must be of {2} type.”, error code to 001 and list of placeholders will contain A1 (name of the attribute), A (name of the dataset) and int (the type of the attribute) in this order. The final text of the message with replaced placeholders will be “Attribute A1 in dataset A must be of int type.”.

Because of the placeholders, we had to add two validations to application descriptor already listed in section 2.5.

1. Dataset name cannot contain {number}—number in curly braces.
2. Value of the name property of an attribute cannot contain {number}—number in curly braces.

Without these validations, some parts of names of datasets or attributes could be replaced during replacements of the message placeholders.

One of our requirements was also multilingual support—requirement R8 (1.3). Because of limited resources, we have decided to prepare all necessary parts for later implementation, but as part of this thesis implement only support for the English language.

## 3.7 Application descriptor format

The structure of the application descriptor was already in detail elaborated in chapter 2. In this subsection, we will focus on selecting the format of the application descriptor. Since the application descriptor will be stored in the data storage of the application generating software or sent to the client

applications, so they know the structure of the application, the size of the application descriptor should be minimized.

The format of the application descriptor must be displayable on various platforms and also support for this format must be widespread between different programming languages. Both of these are required so developers for various platforms can work with the descriptor and create MetaRMS client applications. Because of this, the requirement R2 (1.3) for MetaRMS to be multiplatform would not be broken. It is also important, that the format is human-readable since one of our requirements—requirement R4 (1.3) was to provide an easy way how to define the application.

By the requirements mentioned above, we are looking for a data serialization format. Because there is a large number of such formats, we have focused only on the most frequently used ones, since these popular formats are widely supported in different programming languages.

**XML** XML is a verbose serialization format. It uses tags—markup constructs—to structure the data. For XML validation XML Schema can be used and also since XML creates a tree structure tree-traversal API DOM can be used for document traversing.

**JSON** JSON is a serialization format, that uses name-value pairs. It supports basic types, such as numbers, strings, booleans, null and also arrays and objects. JSON Schema ([json-schema.org](http://json-schema.org)) can be used for validation.

**YAML** YAML is human-readable serialization format. This format uses whitespace indentation and leading hyphens for easier readability. The advantage over JSON is support for comments, extensible data types and mapping types preserving key order. On YAML homepage ([yaml.org](http://yaml.org)) Rx and Kwalify are recommended for schema validations, but none of them have support for C# language, which we have selected in section 3.2.

All of the above-mentioned formats are supported by many programming languages. If ordered by verbosity, XML is the most verbose, followed by JSON and YAML as the least verbose. YAML supports some features over JSON, though we have not found these features applicable for application descriptor definition. Even though YAML would save us the most space in the database as well as the bandwidth, we have opted for JSON. The first reason is that it provides an easy schema for validation and the second is, that we are already using API in JSON format as defined in section 3.1, and opting for JSON would not add requirements to another support libraries.

### **3.7.1 Application descriptor validation**

As mentioned in section 2.5, when selecting an application descriptor format, it is important to be able to perform validations, preferably against a schema. Selected format JSON provides such with the JSON Schema, but even this validation against schema cannot perform all the necessary validations of the application descriptor mentioned in the section 2.5. Because of this some of the validations will have to be performed in the code before the application creation. In this subsection, we will provide a list of validations necessary to be performed on the application descriptor, divided to validations performed with the JSON Schema and in the code.

#### **Validations available with JSON Schema**

Since JSON provides some validations using JSON Schema, we have decided to use it for validation of the application descriptor structure. Via the JSON Schema, the following validations are performed:

1. Descriptor structure and its required fields (ApplicationName, LoginApplicationName, DefaultLanguage, SystemDatasets, Datasets)
2. Validation that no other than valid fields are present in the descriptor
3. Types of properties values, such as
  - (a) ApplicationName, LoginApplicationName, etc. are strings
  - (b) Min, Max, etc. are numbers
  - (c) Required, Safer, etc. are booleans
  - (d) DefaultLanguage, OnDeleteAction are values from enumerations
4. That there is at least 1 User-defined dataset

#### **Validations that need to be done in the code**

All the validations from section 2.5 not mentioned already in validations available with the JSON Schema, as well as two additional validations from section 3.6 will be performed in the code. These validations either cannot be performed via the JSON Schema at all or their expression would be too complicated, which would cause worse readability of the schema.

### 3.7.2 Default values setting

At the complete list of attribute properties in subsection 2.3.1, we have marked some of the attributes properties as optional and some as required. Because of this, some properties may be missing from the application descriptor. These missing properties will have in JSON a null value assigned automatically. This is not always what we want since the null values should be present only for properties not valid for the actual attribute. To prevent this situation from happening, we have to assign some of the values programmatically. This includes setting:

- the required property to true if the min property is set,
- the min property to 1 if the required property is set to true,
- the required property to false if it is not set,
- the unique property to false if it is not set,
- the safer property of the Password attribute to false if it is not set.

On top of that, we have decided to assign each dataset a unique number within the application that will be used for storing data of user rights sets defined in section 3.5.

- -1 for System users dataset
- Unique number starting from 1 to each User-defined dataset

Because of this rights can now be stored as key-value pairs with dataset id as key and level of the rights, defined in the Representation subsubsection of section 3.5, as value.

### 3.7.3 Final application descriptor structure in JSON

This subsection contains the final structure of the application descriptor in JSON format with an explanation of each field and examples of valid values. This structure is based on the structure elaborated in chapter 2 and is written in JSON format selected previously in this section. Examples of real application descriptors can be found in section 5.3.

```
1 // This is a basic structure of application descriptor
2 {
3     "ApplicationName": "name of the application displayed after login",
4     "LoginApplicationName": "name of the application used for login, this name
5         must be unique between all applications",
6     "DefaultLanguage": "default language - currently only en is supported",
7     "SystemDatasets": {
```

```

7   "UsersDatasetDescriptor": {
8     "Name": "name of System users dataset",
9     "Description": "optional description of System users dataset",
10    "PasswordAttribute": {
11      "Name": "name of password attribute",
12      "Description": "optional password description - useful especially if
13          safer is set to true",
14      "Type": "password",
15      "Required": true,
16      "Safer": "optional boolean value, true means that the password must
17          be at least 8 characters long, must contain lower- and upper-
18          case character and number, if this value is not present false is
19          filled automatically",
20      "Min": "optional positive integer number value of minimal password
21          length",
22      "Max": "optional positive integer number value of maximal password
23          length"
24    },
25    "Attributes": [
26      {
27        "Name": "name of the username attribute",
28        "Description": "optional description of the username attribute",
29        "Type": "username",
30        "Required": true,
31        "Unique": true,
32        "Min": "optional positive integer number value of minimal username
33          length",
34        "Max": "optional positive integer number value of maximal username
35          length"
36      }
37    ]
38  },
39  // Other System users dataset attributes ... - the structure is the
40  // same as for attributes in the User-defined datasets described
41  // below
42  ],
43 },
44 // User-defined datasets
45 "Datasets": [
46  {
47    "Name": "dataset name - this must be unique between application
48      datasets",
49    "Description": "optional dataset description",
50    "Attributes": [
51      {
52        // Example of basic data attribute
53        "Name": "name of the attribute - this must be unique within
54          dataset attributes",
55        "Description": "optional attribute description",
56        "Type": "type of attribute data, available values for basic type
57          are color, date, datetime, email, month, int, float, year,
58          phone, string, time, url, bool, text",
59        "Required": "optional boolean value, if set to true, the attribute
60          is required to be filled, if this value is not present false
61          is filled automatically",
62        "Min": "optional integer number value that means for numeric types
63          (int, float, year) minimum value and for text types (text,
64          string, username, password) means minimal string length",
65        "Max": "optional integer number value that means for numeric types
66          (int, float, year) maximum value and for text types (text,
67          string, username, password) means maximal string length"
68      },
69    ]
70  }
71 }

```

```

49     // Example of attribute of type reference
50     "Name": "name of the attribute - this must be unique within
51         dataset attributes",
52     "Description": "optional attribute description",
53     "Type": "type of reference attribute must be SystemDatasets.
54         UsersDatasetDescriptor.Name or correspond to a name of any
55         User-defined dataset in the application descriptor",
56     "Required": "optional boolean value, if set to true, the attribute
57         is required to be filled, if this value is not present false
58         is filled automatically",
59     "Min": "optional positive integer number value that means minimum
60         of references that the attribute must contain",
61     "Max": "optional positive integer number value that means maximum
62         of references that the attribute can contain",
63     "OnDeleteAction": "possible values are cascade, setEmpty, protect"
64   }
65   // Other dataset attributes ...
66 ]
67 }
68 // Other User-defined datasets ...
69 ]
70 }

```

## 3.8 Data layer

The data storage located on the server side of MetaRMS (briefly introduced in section 3.1) is an important part of the software architecture. It was also already mentioned, that this storage will be implemented as a relational database, that has to be able to store login credentials, information about generated applications and applications' data. In this section we will elaborate the whole data layer in more detail, focussing on the approach of storing the data into the database, selection of the database itself and also on a way of accessing the database from the code.

### 3.8.1 Database schema

Before the actual selection of the database, we need to design a database schema. MetaRMS will need to hold data for multiple applications and the structure of the data will be based on the structure defined in the application descriptors. Because of that, we will not know the structure of stored data when developing and deploying MetaRMS. For storing the data of unknown structure, we have thought of the following approaches:

1. When creating a new application based on the application descriptor we will create a database table for each dataset from the descriptor. The columns of the table will correspond to the dataset attributes. With this approach, the database would be modified on every creation of a new application and it will soon become overwhelmed by tables.

2. Store data of all datasets in a single table. This table will contain a column for an application identifier, dataset identifier, attribute identifier, record identifier, and type identifier and columns for each Basic type and one column for a Reference type. One row in this table will represent value stored in one attribute so one record will be spread through multiple rows. With this approach, every row will have only one of the type columns filled, based on the value in the column of type identifier. This results in many null values in the table and we have also found this concept to be too complicated. This approach was inspired by Single table strategy for solving inheritance, explained in the article [Pra14], but it was strongly modified to suit our needs.
3. Store data in a serialized form. This approach provides us with the possibility to store data of all datasets in a single table by serializing them into one column. For this approach, each row will contain an identifier of application and dataset and a column with serialized data of the record. The format of the serialization can be for example XML or JSON since there are databases with support for columns of these types. From our point of view, we have found storing data in JSON as the most suitable option, because JSON is already used for representing application descriptor as selected in section 3.7.

From the above-mentioned ideas, we have opted for approach 3 since this approach makes the database well-arranged. This approach does not require the database to be changed programmatically—we will not need to add more tables—and it also does not leave any empty columns. This selected approach will be adopted for storing application data, such as records of individual User-defined datasets and System users dataset. Also storing rights sets will be implemented this way since the datasets of each application will differ.

When storing data of records of the User-defined datasets and the System users dataset, each attribute can have one or more values. For every attribute with the type property set to the Basic type, only one value would be valid—be it one number, one string or one date. However, for the Reference type attributes, the number of values depends on the min and max properties of the attribute—each value represents one reference. In order to keep the serialized structure uniform, it needs to contain a dictionary with the attribute being the key and list of values being the value—this way values of both Basic and Reference type attributes can be stored. In C# this structure can be represented by `Dictionary<string, List<object>>`. An example of such JSON structure is: `"A1": [1, 2], "A2": ["hello"]`—A1 is a Reference type attribute with references to records with identifiers 1 and 2 and A2 is a Basic type attribute (string or text for example).

There are four main components that need to be stored in the database:

- Generated applications
- Rights sets
- Users
- Application data

This structure is in the figure 3.3 and description of each component with more details will follow.

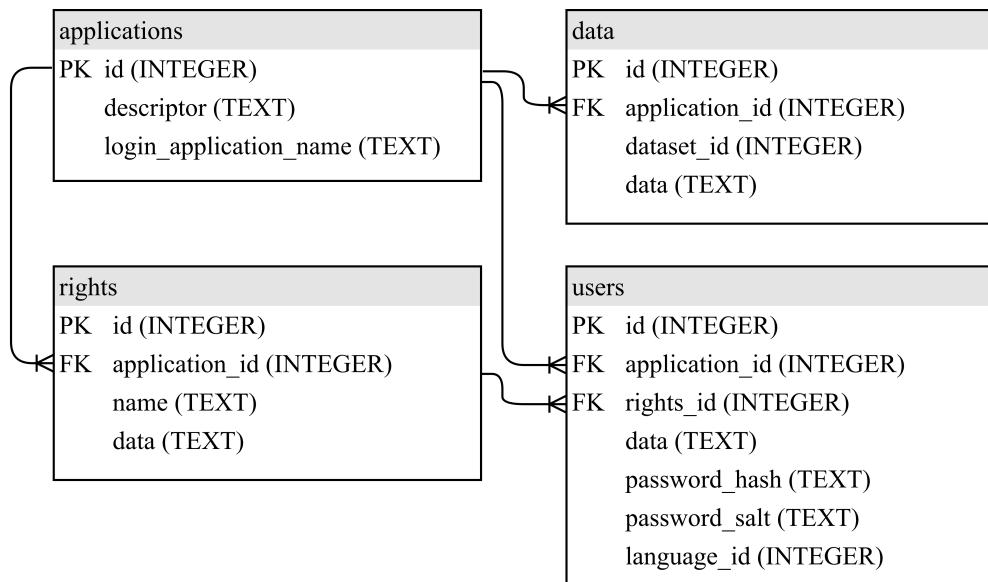


Figure 3.3: MetaRMS database schema.

Now let us elaborate each component of the schema in figure 3.3 separately in more detail:

**Applications** It is obvious, that we need to save the applications somewhere. For each application we need to store its application descriptor, which contains the application structure as described in chapter 2 and can be used to create the application.

- Application descriptor – in JSON format as decided in section 3.7 and validated as stated in subsection 3.7.1.
- Login application name – in the Login credentials subsubsection of section 3.4 we have decided to use a unique application

name together with username and password to authenticate a user. Because of this for every attempt to log in we would need to find the correct application descriptor and get the LoginApplicationName from it. The more applications will be within MetaRMS, the more demanding will this task become. Because of this, we have decided to duplicate the LoginApplicationName and store it both in the application descriptor and as additional information about the application for easier querying. Even though this creates duplicities within the database, since the LoginApplicationName cannot be changed, this duplicity is not a problem.

**Rights sets** As mentioned at the beginning of section 3.5 users will have a set of rights assigned. These sets need to be stored in the database so users can have them assigned. As mentioned in the Granularity subsubsection of section 3.5 rights for each dataset need to be stored and for each dataset it will be one value, defined in the Representation subsubsection of section 3.5. For each set of rights we need to store:

- Name (of the rights set) – As briefly mentioned in section 3.5 each rights set will have a name assigned. This way when selecting a rights set for a user it could be easily found by its name. For example in a library from our representative scenarios 2a we could have a rights set named “librarian” and when creating a new user on a librarian position it will be easy to find the correct rights set to assign.
- Data (of the rights set) – As already defined in subsection 3.7.2 each dataset is assigned a unique number. These numbers together with the rights values, elaborated in the Representation subsubsection of section 3.5, create a key-value pair—the key is unique dataset id and the value is the level of the rights. A single rights set then contains exactly one pair for each User-defined dataset, one pair for System users dataset (with key -1, as decided in subsection 3.7.2) and one pair for rights sets. Even though rights sets are not called a dataset, in some cases, they behave similarly and because of that, each user needs to have rights to them assigned. Because of that rights sets will also have a unique number (the key) assigned—since System users dataset has number -1, we have decided that rights sets will have number -2 assigned.
- Application identifier – Each rights set belongs to exactly one

application. We need to be able to link the rights to an application and because of it, we need to store the reference to the application. This value is a reference to the *Applications table*, previously mentioned in this subsection 3.8.1.

**Users** Users table will contain all users of the MetaRMS applications. This table will be used for authentication of the users and because of that it needs to contain the login credentials as mentioned in the Login credentials subsubsection of section 3.4. Also based on the application descriptor each user can have additional information, that will also be stored within this table. The following enumeration lists all columns within the table.

- Data (about the user) – Every user needs to have at least username, but can also have another information filled. The structure of these pieces of information is based on the application descriptor, more specifically on the attributes in the System users dataset, as defined in section 2.2. As decided at the beginning of this subsection, these pieces of information will be stored as a serialized JSON dictionary.
- Password hash – As defined in the Password storing subsubsection of section 3.4 we have decided to store the password in a hashed form.
- Password salt – As mentioned in the Password storing subsubsection of section 3.4 to enhance security we have decided to use salt together with the password and for successful check of the password correctness, the salt needs to be stored as well.
- Rights set identifier – As defined in section 3.5 every user will have a set of rights assigned. Because of this, we need to store the identifier of the rights set belonging to the user. This value is a reference to the *Rights sets table*, previously mentioned in this subsection 3.8.1.
- Application identifier – Each user belongs to exactly one application. We need to be able to link the user to an application and because of it, we need to store the reference to the application. This value is a reference to the *Applications table*, previously mentioned in this subsection 3.8.1.
- Language – To support multiple languages—as required by the requirement R8 (1.3)—every user has a language assigned. This language can be used for the user interface language and for

the language of messages received from the server. As already mentioned in section 2.1 because of the limited resources only English will be supported.

**Data (of the generated applications)** Data table will contain data from all datasets of all applications. Because of that, each record needs to have an identifier of the application and the dataset it belongs to. Since datasets can have different structures, storing the data as serialized objects was selected at the beginning of this section. All of the stored information are listed in the following enumeration.

- Data – This column will contain data of one record for a dataset. As decided at the beginning of this section, these pieces of information will be stored as a serialized JSON dictionary.
- Dataset identifier – Identifier of the dataset the record belongs to. As mentioned in subsection 3.7.2 each dataset has a unique id within the application assigned and this column will contain this id.
- Application identifier – Each data belongs to exactly one application. We need to be able to link the data to the application and because of it, we need to store the reference to the application. This value is a reference to the *Applications table*, previously mentioned in this subsection 3.8.1.

### 3.8.2 Data storage choice

The database schema from figure 3.3 is quite simple, thus it does not limit us in selecting the database. Our main requirement is support in .NET Core (selected as the development platform in section 3.2) and the requirement R2 (1.3) for MetaRMS to be multiplatform. The interesting part is storing JSON formatted data as defined in subsection 3.8.1. Because of these columns we were also considering databases with JSON columns support. From our requirements we were deciding between:

- PostgreSQL – this is an advanced relational database with a wide variety of supported data types, including JSON. PostgreSQL database is great for complex operations and larger projects.
- MySQL – this is a very popular open-source database that offers support for JSON column type. This database needs a MySQL server to run and is suitable for distributed web applications.

- SQLite – this is a simple free and open-source database, that runs directly on the filesystem. Because it does not need a server to run, no configuration is needed to start using it. Even though the default supported types do not include the JSON data type, after additional research we have found that a JSON1 extension pack exists, which adds JSON support to the SQLite database. More information about this extension pack can be found on the official site [www.sqlite.org/json1.html](http://www.sqlite.org/json1.html).

As a helpful source of information about different databases has proven the article [Mar19]. Even though both PostgreSQL and MySQL support JSON queries by default, we have decided to benefit from the SQLite database and its option to run directly on the filesystem. This option creates fewer requirements on the hosting, that will be selected in section 3.9. Also even though JSON querying will not be a part of this thesis, later when extending the code with this functionality, developers can benefit from the possibility of using the above-mentioned extension. When SQLite will not be enough to handle requests, MySQL would make a great replacement.

### 3.8.3 Accessing the database from the code

The above selected SQLite database will be accessed from the C# code of the Server application mentioned in section 3.1. To make this access easier for us we have decided to implement it using an Object-Relational mapping—so-called ORM. For .NET Core we were selecting from the following options:

- Entity Framework Core is an open-source cross-platform version of Entity Framework, an ORM framework from Microsoft. This framework allows Code-First and Database-First approaches and removes the need for writing raw SQL queries into the code by supporting LINQ.
- Dapper is a so-called micro ORM framework, that was created with performance in mind. Because of that developers write queries in raw SQL and thus can fully influence how the query looks.

When selecting between the ORM frameworks we have considered experiences from the [Cin17] bachelor thesis. In this thesis in section 4.7.2 on page 37, the author evaluates the selection of Dapper as a not good choice. The main reason is, that raw SQL queries in the code are not flexible to debug and edit. Because of that, we have opted for Entity Framework Core.

With Entity Framework Core selected as an ORM framework, we can choose between the Code-First and the Database-First approach of Data

Modeling. For this we have found the article [Rya18] helpful. To mention the important differences between each of the approaches:

- In Code-First approach, the developer can define data model objects using standard classes and the ORM framework then generates the database base on them.
- Database-First approach is suitable in situations where a database is already created and it needs to be mapped to classes. This way the ORM framework takes the database and generates model classes automatically.

The Code-First approach has been selected to be implemented since we would like to avoid the automatic generation of the code. This approach allows us to have full control over the code and perform changes in the database by changing the code first and then make a database migration.

## 3.9 Hosting

To test MetaRMS in the production environment, we have decided to deploy it to the server accessible in the Internet. For this deployment to be available to anybody, we have decided to prefer a free hosting. Since we have already made the decision about the language and platform in section 3.2 used for MetaRMS development, we can list requirements on the server which we will deploy to.

### **SR1** free hosting

Selection of ASP.NET Core as a platform makes the second requirement to be the support of this platform and since we are developing MetaRMS on ASP.NET Core 2.0 this is the exact required version. It would be also preferred that the server will have a support of the future versions of ASP.NET Core for future migration to the newer versions.

### **SR2** support of ASP.NET Core 2.0

### **SR3** support of future versions of ASP.NET Core

We have also already decided about the database used as a data storage. Since the choice was in subsection 3.8.2 SQLite database, that does not require any database server and it is just a single file, we have no requirement on the database server.

All of the above-mentioned requirements were satisfied by [www.aspify.com](http://www.aspify.com) and their free web hosting. The advantage of this web hosting is that in the future, we can change the plan to the paid one with additional features.



# 4. Implementation

This chapter contains a description of MetaRMS implementation. The implementation is based on observations and decisions from chapter 3. With information from this chapter a person with a programming background should understand what are the individual parts of the project and in case of implementing a new feature, the developer should be able to find the part of the code that needs to be changed.

The structure is based on the software architecture elaborated in section 3.1. Before describing the individual parts of the MetaRMS solution in detail, we will introduce basic workflow and requirements, as well as individual processes available within MetaRMS.

## 4.1 Basic workflow and requirements

The figure 4.1 contains a simplified workflow of MetaRMS. It also denotes that MetaRMS is divided into two projects—**Core** and **SharedLibrary**. The following list contains an explanation of the individual parts of the workflow.

**Administrators/Users** End users interacting with the Web client selected in section 3.1. This can be either administrator creating a new application or a regular user logging into an application and performing authorized CRUD operations on the data.

- 1 The interaction between end-user and Web client is performed via a web browser available on both desktop and mobile devices.

**Razor web client** Web client is implemented in ASP.NET Core 2.0 with the help of Razor Pages selected in section 3.2. This client provides web pages accessible via web browser and connects to the MetaRMS server JSON API endpoints.

- 2 Both the Web client and MetaRMS server are using methods from the **SharedLibrary**. These methods are invoked by C# method calls.

**SharedLibrary** The **SharedLibrary** is implemented in C# as .NET Standard 2.0 library. This library is used by both Web client and MetaRMS server.

- 3 JSON web API is used by the client applications to connect to the MetaRMS server. This API was selected in section 3.1. When a

user is signed, JWT authentication token is sent to the server with every request as decided in the Authentication method subsubsection of section 3.4.

**Controllers** Controllers contain API endpoints to which the client applications are connecting. Controllers are part of the `Core` project, which is implemented in ASP.NET Core 2.0.

**Repositories** Repositories are the access points to the database. Repositories are part of the `Core` project, which is implemented in ASP.NET Core 2.0.

4 Entity Framework Core 2.0 picked in subsection 3.8.3 is implemented as an object-relational mapper between MetaRMS server and the database.

**SQLite database** SQLite database was selected as data storage in subsection 3.8.2. This database is located within the `Core` project.

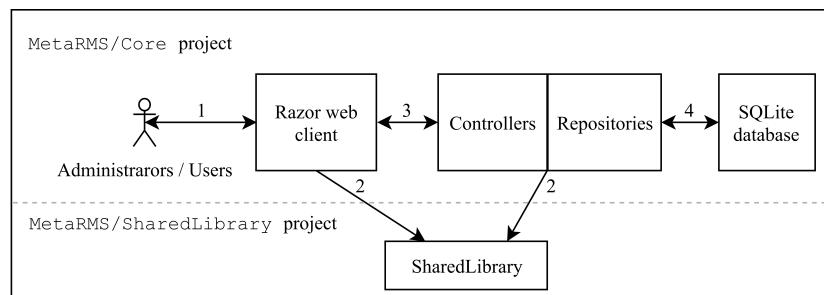


Figure 4.1: Basic workflow of MetaRMS solution.

In the following section 4.2 this workflow will be elaborated for individual processes within MetaRMS in more detail.

## 4.2 MetaRMS processes

Before explaining individual parts of MetaRMS solution it is important to elaborate processes available within. By process, we mean a sequence of steps leading to a certain result. Each of the processes represents an action that a user can perform in the application. In each of the following subsections, one of the processes will be explained in detail. Every figure in the following subsections is also described in detail and labels from the figures are written in *italics* to be easily distinguishable from the other text. Web paths in this section are relative to the location of the Web client application home address.

### 4.2.1 Application initialization

The process of application initialization or in other words the process of creating a new application is described in the figure 4.2. In this process administrator provides an application descriptor as defined in the chapter 2 and section 3.7 and an email address to the application initialization page accessible via a web browser on path `/AppInit`. This page in the code corresponds with the `AppInit/Index.cshtml` and `AppInit/Index.cshtml.cs` files in the `Pages` folder of the `Core` project. The application descriptor together with the email address is then sent via the JSON web API (in the figure represented as 1) to the `AppInitController` on the server side. This controller performs validations—both against the JSON schema and in-code—mentioned in subsection 3.7.1. In case of an invalid application descriptor or email address, error messages (described in section 3.6) are sent back to the client application (represented by 8 in the figure).

If no errors were found, a new application based on the application descriptor is created by calling the `ApplicationRepository` functions (represented by 2). This new application is stored to the *Database* using Entity Framework Core 2.0 (represented by 3). After the application is created an administrator account needs to be set up. Before that, administrator rights set needs to be made. This is performed by calling the `RightsRepository` functions (represented by 4) that store the rights set into the *Database* again using Entity Framework Core (represented by 5). Rights for the administrator are set to full CRUD for all User-defined datasets, System users dataset and rights sets—allowing the administrator to perform all operations (explained in the Representation subsubsection of section 3.5). The administrator user account is created using the `UserRepository` (represented by 6) with the rights set to the newly created rights and the username set to “admin”. The new user entity is stored into the *Database* by Entity Framework Core (represented by 7).

After the application, administrator rights set and account are created, an email with administrator login credentials is sent to the provided email address. If everything is successful a message about the success is returned back to the web client application (represented by 8). The client then performs redirect to the login page (represented by 9) located on the `/` path. This page is in the code represented by the `Index.cshtml` and `Index.cshtml.cs` files in the `Pages` folder of the `Core` project. After the email is received administrator can log into the new application.

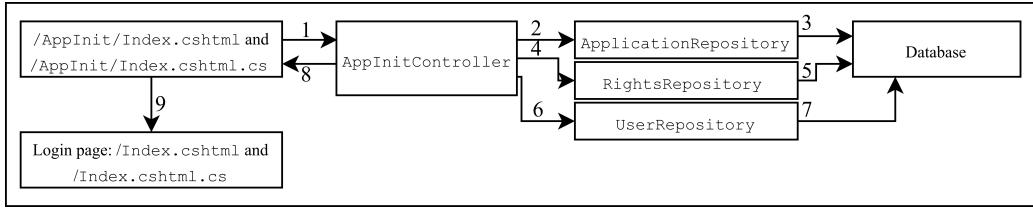


Figure 4.2: Application initialization process.

#### 4.2.2 Authentication

This subsection contains process of logging into and out of the application described in more detail.

##### Login

The login process starts on the login page and is described in the figure 4.3. This page is located on the / path and is represented by the `Index.cshtml` and `Index.cshtml.cs` files in the `Pages` folder of the `Core` project. User needs to provide the unique application name, username and password as authentication credentials, as decided in the Login credentials subsubsection of section 3.4. These credentials are sent via the JSON web API (represented by 1) to the `LoginController` on the MetaRMS server. In this controller, the credentials are checked with the help of the `UserRepository` (represented by 2) against the `Database` via Entity Framework Core 2.0 (represented by 3). If credentials are not valid an error message (described in section 3.6) is returned back to the client (represented by 4) and displayed on the page.

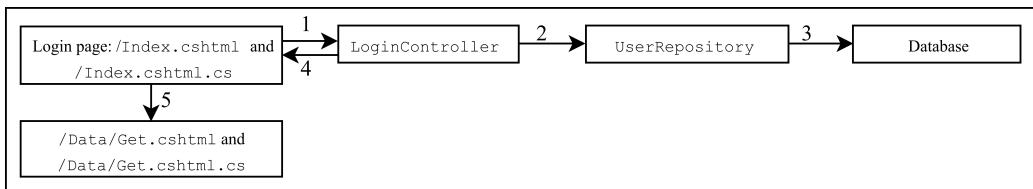


Figure 4.3: Login process.

In case of successful login the `LoginController` creates a JWT authentication token (described in the Authentication method subsubsection of section 3.4) that is returned back to the client application (represented by 4). The client application then performs redirect from the login page to the dataset data page containing data of the first dataset that the logged user has rights to see or, in case no such dataset exists, to a page with no data

(represented by 5). The dataset data page is on the `/Data/Get/<dataset name>` path and is represented in the code by the `Data/Get.cshtml` and `Data/Get.cshtml.cs` files in the `Pages` folder of the `Core` project.

## Logout

The logout process is displayed in the figure 4.4. The logout page is accessible via Logout button displayed on the pages visible for logged users. This page is on the path `/Account/Logout` and is in code represented by the `/Account/Logout.cshtml` and `/Account/Logout.cshtml.cs` files in the `Pages` folder of the `Core` project. When user is redirected to the logout page the `LogoutController` is called via the JSON web API (represented by 1). The current version of the controller only returns HTTP 200—OK—response (represented by 2). After that the logout page deletes cookies and performs redirect to the login page (represented by 3).

This simple implementation of logout on the server in the `LogoutController` was selected due to limited resources. In the future, the logout process on the server side should be reimplemented. For example, we could have a database of no longer active tokens that still have not expired. Then with every authorized request, we would query the provided token against this database and allow access only to valid tokens not contained in this database.

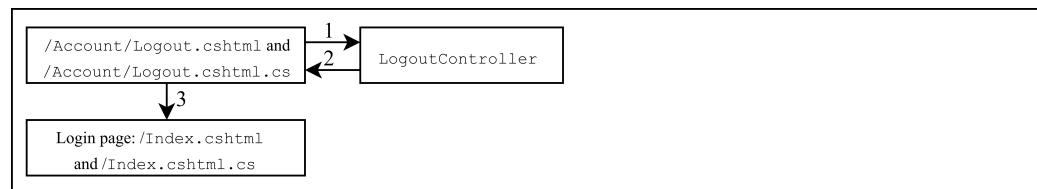


Figure 4.4: Logout process.

### 4.2.3 Operations

Figure 4.5 contains processes of read, create, edit and delete operations that are common for System users dataset, all User-defined datasets, and rights sets. In the figure, there is a \* symbol, used as a placeholder for words User, Data and Rights representing System users dataset, User-defined datasets, and rights sets respectively. For each of them, the implementation of individual operations will be elaborated in more detail under the figure. System users dataset also contains one additional operation—resetting a password—which will be in more detail elaborated in the System users dataset subsubsection.

Before listing all the operations it is important to mention some parts of the processes that are common to all.

- All of these operations can be performed only by a user who is logged in and owns a valid JWT authentication token (process of receiving the token was described in the Login subsubsection of subsection 4.2.2). This token is sent with every request from the client application to the server, where the token is validated.
- Before every operation, the logged user is checked whether they are authorized for the operation. The result of the authorization is based on the user's rights set as described in section 3.5.
- All communication between repositories and the *Database* is done with Entity Framework Core 2.0 selected in subsection 3.8.3 (represented in the figure by 18 and 19).
- All the page files—the .cshtml and .cshtml.cs files—are located in the **Core** project in the Pages/\* subfolder. For example files regarding System users dataset are located in the Pages/User folder.
- All controllers are located in the **Core.Controllers.\*** namespace. For example controllers regarding System users dataset are located in the **Core.Controllers.User** namespace.

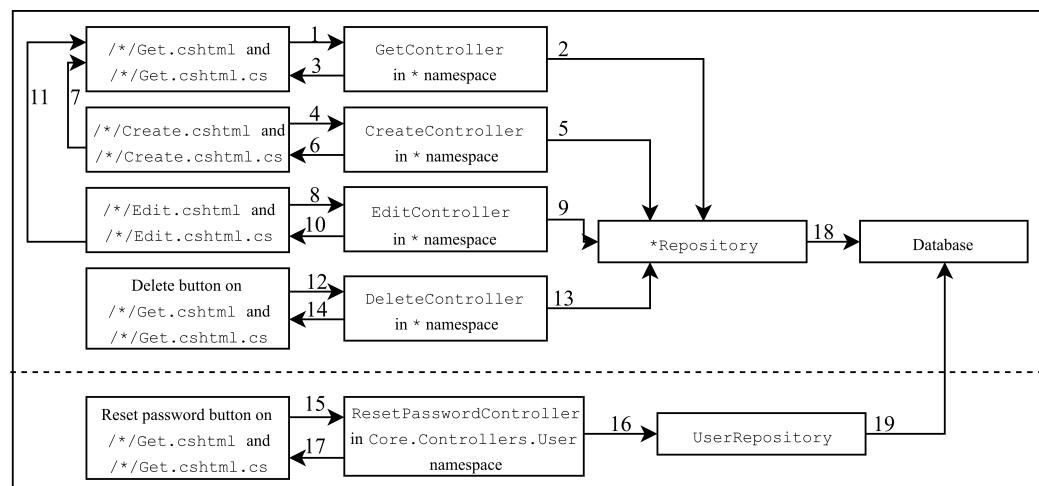


Figure 4.5: Process of CRUD operations on User-defined datasets, System users dataset and rights sets.

## System users dataset

System users dataset was with all the details elaborated in the section 2.2. MetaRMS provides 5 operations on this dataset and the processes of these operations will be explained in the following enumeration. As mentioned at the beginning of this section, for the System users dataset the \* in the figure 4.5 stands for User.

**Get all users of the application** This operation starts when a user sends GET request to the get users page located on the /User/Get path. The page is in the code represented by the /Get.cshtml and /Get.cshtml.cs files. From this page request to the GetController is sent by JSON web API (represented by 1). The GetController calls the UserRepository (represented by 2) and the application users are loaded from the Database. When users are loaded the data is sent back to the client application (represented by 3) and finally displayed.

**Create a new user** The user create page contains the empty structure of a new user—all fields defined in the descriptor of the current application and a select box with the available rights sets for the user. These application descriptor-based fields are taken from the attributes of the System users dataset descriptor. Their type, the fill requirement and the valid values are based on the properties of each attribute. The valid rights sets and valid references for the reference type attributes are loaded from the server with the GET request to the user create page. This page is located on the /User/Create path and is in code represented by the /Create.cshtml and /Create.cshtml.cs files.

After the logged user fills all the necessary information about a new user and clicks on the Save button all the input values are serialized and sent via the JSON web API (represented by 4) to the CreateController on the server. This controller is responsible for input validations based on the application descriptor. In case of errors, the error messages are returned back to the client (represented by 6). If the input was correct using methods from the UserRepository (represented by 5) the new user is serialized and stored in the Database.

After the successful creation, a message is sent to the client application (represented by 6) and the user is redirected to the get users page mentioned in the previous bullet (represented by 7). The new user account is ready immediately. The password for every new user is set to “MetaRMS123” and this default value can be changed in the SharedLibrary in file with constants.

**Edit an existing user** All information about application user—except password—can be edited on the user edit page. This page is similar to the user create page mentioned in the previous bullet. The difference is that the user edit page contains already filled information about a user to edit. Also the edit page is located on the `/User/Edit/<id of the user to be edited>` path and is in the code represented by the `Edit.cshtml` and `Edit.cshtml.cs` files.

The rest of the process is very similar as well. Modified values are sent to the `EditController` (represented by 8), validated and stored in the `Database` using the `UserRepository` (represented by 9). Messages handling is identical to the one in creation of a new user and is represented by 10. In case of successful edit, the user is redirected to the get users page mentioned in the first bullet (represented by 11).

**Delete an existing user** The delete operation is invoked by clicking on the *Delete button* associated with a specific record on the get users page mentioned in the first bullet. The request is sent via the JSON web API to the `DeleteController` on the server (represented by 12). If the `Database` restrictions would not be violated by the delete operation, the delete is performed by methods from the `UserRepository` and `DataRepository` (represented by 13). The delete operation might delete other records as well based on the application descriptor settings of the `onDeleteActions` defined in the bullet 7 of subsection 2.3.1. The message about the result is sent back to the client application (represented by 14).

**Reset password of an existing user** The reset password operation is invoked from the get users page mentioned in the first bullet by clicking on the *Reset password button*. Its goal is to reset the password of the selected user back to the default value—“MetaRMS123”. The reset request is sent via the JSON web API (represented by 15) to the `ResetPasswordController`. The new password is stored in the `Database` using the `UserRepository` methods (represented by 16). Message about the action result is then sent back to the client application (represented by 17) and displayed.

## User-defined datasets

When performing operations on the User-defined datasets it is important to know what dataset are we dealing with. Because of that, each of the processes listed below needs to know the dataset identifier. As mentioned

at the beginning of this section, for User-defined datasets the \* in the figure 4.5 stands for Data. Also, some parts of the processes are very similar to the processes already mentioned in the System users dataset subsubsection mentioned above and because of that, we will use it as a reference.

**Get all records from one dataset** This operation is very similar to the get all operation in the System users dataset described in the previous subsubsection. The difference is that it starts by sending the GET request to the data get page on the `/Data/Get/<name of the dataset to get>` path. On the server, the data are loaded using the `DataRepository`.

**Create a new record in a dataset** The create operation on the User-defined datasets is similar to the create operation on the System users dataset. The data create page contains an empty structure for a dataset record and is located on the `/Data/Create/<name of the dataset to create the data to>` path. The structure and the input types are based on the application descriptor and attribute properties of the dataset to which the record belongs. In case of the reference type attributes, the valid references are loaded from the server with the GET request to the data create page.

After the user fills the necessary information about the new record and clicks on the Save button, the input values are serialized and sent via the JSON web API (represented by 4) to the `CreateController` on the server. This controller is—as well as in the case of the System users dataset—responsible for input validations. In case of errors, the error messages are sent back to the client (represented by 6). Otherwise the data are stored in the `Database` with the help of the `DataRepository` (represented by 5) and a message about success is sent back to the client application (represented by 6). Client application redirects the user to the data get page mentioned in the previous bullet (represented by 7) and displays the message.

**Edit existing record in a dataset** The data edit page contains information about single already created record. The page is located on the `/Data/Get/<name of the dataset the record is from>/<id of the record to edit>` path. All the values displayed on this page, as well as valid references for reference type attributes, are loaded from the server with the GET request to the data edit page. When the user clicks the Save button, the edited structure is serialized and sent via the JSON web API to the `EditController` on the client

side (represented by 8). The controller performs validations against the application descriptor and, in case of errors, returns error messages (represented by 10).

If no errors are found the record is stored in the *Database* with the help of the *DataRepository* (represented by 9). Then message about success is sent back to the client application (represented by 10). The client redirects the user to the data get page, mentioned in the first bullet, and displays the message (represented by 11).

**Delete an existing record in a dataset** The delete operation is almost the same as for the System users dataset mentioned in the previous subsubsection. The only difference is that the *Delete button* is located on the data get page. Again it is important to remember that other records might be deleted as well based on application descriptor settings of the `onDeleteActions` defined in the bullet 7 of subsection 2.3.1.

## Rights sets

Rights sets were in detail described in section 3.5 and in this section, we will elaborate available operations that can be performed with rights sets. As mentioned at the beginning of this section, for rights sets the \* in the figure 4.5 stands for Rights.

**Get all rights sets of the application** This operation is similar to the get all operations on the User-defined and the System users datasets. It starts by sending the GET request to the rights get page located on the `/Rights/Get` path. The page then sends a request to the *GetController* (represented by 1). This controller returns all available rights for the application the user is logged in from the *Database* to the client (represented by 3), with the help of *RightsRepository* (represented by 2). The client then displays all the received rights sets.

**Create a new rights set** The creation of a new rights set starts on the rights create page located on the `/Rights/Create` path. This page contains a text input field for the rights set name and a select box with available rights values—None, R, CR, CRU and CRUD (defined in the Representation subsubsection of section 3.5) for each dataset of the application. When the values of the new rights set are filled and the user clicks on the Save button, the input data are serialized and sent to the *CreateController* via the JSON web API.

The controller performs validations mentioned in the Rights validation subsubsection of section 3.5 and in case of errors returns error messages to the client (represented by 6). If the rights are correct, they are stored in the *Database* with the help of the **RightsRepository** (represented by 5). Information about success is sent to the client (represented by 6). The client then performs a redirect to the rights get page, mentioned in the previous bullet, and displays the message.

**Edit an existing rights set** The structure of the rights edit page is the same as the structure of the rights create page mentioned in the previous bullet with the exception that the values are already filled and ready to be modified. Also the edit page is located on the `/Rights/Edit/<id of the rights set to be edited>` path. After the user clicks the Save button to save the modifications, the data are serialized and sent via the JSON web API (represented by 8) to the **EditController** on the server. The rest of the process is the same as for the new rights set creation mentioned in the previous bullet.

**Delete an existing rights set** The delete operation is invoked by clicking on the *Delete button* associated with a specific rights set on the rights get page. The request is sent via the JSON web API to the **DeleteController** on the server (represented by 12). If no user has the given rights assigned, the method in the **RightsRepository** is called (represented by 13), the rights are deleted from the *Database* and a success message is sent back to the client application. Otherwise, an error message is sent back to the client (represented by 14).

#### 4.2.4 Settings

This subsection contains a process regarding account settings for a currently logged user. The implemented process is changing the user password, which will be in more detail described in the following subsubsection.

##### Password

This process relates to the password of the currently logged user and is depicted in the figure 4.6. Each user can change their own password in the settings of the application after login. The settings page is on the `/Account/Settings` path and is in code represented by the `/Account/Settings.cshtml` and `/Account/Settings.cshtml.cs` files in the `Pages` folder of the `Core` project. On the settings page, the user fills the current password and the new password. The new password needs to

be entered twice to prevent possible typing errors. All these information are sent via the JSON web API (represented by 1) to the `PasswordController`.

This controller performs password validations based on the password attribute properties of the application descriptor (explained in section 2.3). If an error is found, the error message is returned back to the client application (represented by 4). Otherwise new password is hashed with salt (explained in the Password storing subsubsection of section 3.4) and stored to the *Database* via the `UserRepository` (represented by 2) using Entity Framework Core 2.0 (represented by 3). After that, a success message is returned back to the client application (represented by 4).

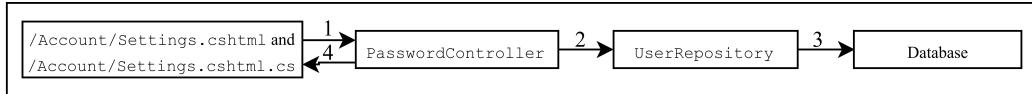


Figure 4.6: Process of changing password of the logged user.

## 4.3 Solution structure

As already mentioned in section 4.1 MetaRMS solution consists of two projects—`Core` and `SharedLibrary`. In the following section, we will annotate the structure of both of the projects to explain the contents of individual parts.

- `Core` project containing server part of MetaRMS—application server and data storage selected in subsection 3.8.2—and Web client since they were merged together as mentioned at the end of section 3.1.
- `SharedLibrary`, a library that provides functions, structures and other helper files shared between the server application and client applications.

Both of these projects will be elaborated in more detail in the following subsections.

### 4.3.1 Core

As decided in section 3.1 the `Core` project contains both the Application server and the Web client application. The structure of this project is divided into folders with common content and the following enumeration describes all the folders of the `Core` project in more detail and when necessary the files are also described.

**Cache** This folder contains class responsible for creating the text representation of the references and stores them in the cache to reduce the number of queries to the database.

**Controllers** This folder contains API controllers. These controllers are expecting HTTP requests on the `/api/<file without Controller name within Controllers`

`folder>` route — for example `Controllers/Data/GetController` is on the route `/api/data/get`. Controllers in this folder are divided into `Data`, `Rights` and `User` subfolders contain controllers for getting, creating, updating and deleting entities from corresponding database tables. The `User` folder also contains controller for resetting a password of an arbitrary user. The `Account` folder contains controllers regarding the account of the currently logged user with methods for login, logout, getting user's rights set and descriptor of application the user is logged in. The `Settings` subfolder contains controller for changing the user's password. The only controller without its own folder is the `AppInitController`, that serves to create a new application.

The general workflow of controllers contains authentication (except controller for login and initialization of a new application), authorization of the user, input data validation (for controllers that create or update the data) and return of the JSON response back to the client application.

In case of creating new API endpoints or updating the existing ones, general changes should be performed within this folder. In case of more specific changes, such as changes in data validations, corresponding helpers should be edited.

**Helpers** This folder contains helper classes for both the Server application and the Web client application. Since every helper serves a special purpose, we will elaborate on each of them individually.

- `AccessHelper.cs` – This helper contains methods necessary for the Web client, used for getting access to the page content. Methods for getting the authentication token, logged user's application descriptor and rights and preparation of the page data can be found here.
- `AppInitHelper.cs` – This helper is used by the Server application and contains methods necessary during the initialization of a new application, such as preparation of administrator's account and sending a confirmation email of successfully created application.

- **CacheHelper.cs** – This helper contains methods for getting application descriptor and rights set of the currently logged user from the cache provided by the Web client application.
- **ControllerHelper.cs** – This helper is used on the server side by the Server application in the controllers. It contains methods, that are used within more controllers or are too large to be part of the controller. This includes methods for authentication, validation of references and deletion of data or users.
- **DataHelper.cs** – This helper serves controllers when preparing data for the clients—it enhances the references with their text representations.
- **HTMLSelectHelper.cs** – This helper is used by the Web client application and it prepares values for HTML selects—it loads all the valid values that can be selected for each select element.
- **PasswordHelper.cs** – This helper contains methods for working with passwords, such as its hashing and validation necessary for the Server application.
- **ValidationHelper.cs** – This helper is used by the Web client application and it validates the structure of the data before sending it to the server.

**Pages** This folder contains all the source codes of the Web client web pages. Every page consists of two files—`.cshtml` and `.cshtml.cs` file with the same name. The `.cshtml.cs` file serves as the code-behind file for the corresponding `.cshtml` file. It loads data from the server via API, creates the model of the page—a structure of data, that is available in the `.cshtml` file—and can perform some validations, such as whether the user has authentication token from the server, is authorized or some data validations before sending the data to the server. The `.cshtml` file contains HTML tags enhanced with Razor syntax that provides C# functions and variables within the file.

In the root of the **Pages** folder, files for About and Login pages are located as well as the main Error landing page. The structure of the subfolders corresponds to the database structure—**User**, **Rights** and **Data** folders contain pages for getting, creating and editing corresponding database entities. The **Account** folder contains pages regarding logged user's account and **AppInit** folder contains a page for a new application initialization. **Shared** folder contains partials—parts of HTML and Razor code—that can be used by the other pages.

Partials for menus and displaying of messages are located here. Also `_GetBuilderPartial.cshtml` and `_InputBuilderPartial.cshtml` partials are in this folder. The `_GetBuilderPartial.cshtml` is used for displaying values for the attributes for individual dataset records. The `_InputBuilderPartial.cshtml`, on the other hand, creates input fields based on attribute properties.

If the web pages should be modified, changes will be performed within this folder.

**Repositories** This folder contains database repositories. Repositories are the only entry point to the database. The `BaseRepository.cs` contains methods for all the other repositories that inherits from this base repository. Each of the rest of the repositories corresponds to one database table and contains methods for accessing the table.

Classes in this folder should be modified in case of adding new methods and filters for getting data from the database.

**Structures** This folder contains structures used within the pages in the `Pages` folder.

**wwwroot** This folder contains CSS and JavaScript files for the Web client and its icon. The `css` folder contains custom styles and the `js` folder can in the future contain custom JavaScript functions. In case of making changes to the style of the pages, only files in `css` and `js` folders should be changed. Folder `lib` contains Bootstrap and JQuery libraries and should not be modified.

**Model.cs** This file contains the database context used within the application to connect to the database.

**Program.cs** This file is the start point of the application.

**Startup.cs** This is the configuration file in which all the services are added.

**database.db** This is the SQLite database file used for storing all the application and user's data.

### 4.3.2 SharedLibrary

Within this thesis, the `SharedLibrary` project is used by both the Server application and the Web client application. In the future, this library should be used by other client applications as well, since methods that might be useful for both server and all the clients are included within it.

The structure of this library is divided into folders with common content. In the following enumeration, we will elaborate on each of the folders in more detail.

**Descriptors** This folder contains individual parts of application descriptor divided into C# classes and is used whenever any of server or client applications need to work with deserialized application descriptor.

**Enums** This folder contains enumerations of valid values for different elements, such as supported languages, types of messages mentioned in section 3.6, onDeleteActions elaborated in the bullet 7 of subsection 2.3.1, rights elaborated in the Representation subsubsection of section 3.5 and identifiers of system datasets. It also contains `AttributeTypeEnum.cs` file which is not an enum, but a List with all the basic attribute types listed in the bullet 3 of subsection 2.3.1 together with the password and the username type. This can be used for validation whether is an attribute of basic or reference type.

**Files** This folder contains files unnecessary to the functionality of the `SharedLibrary`. These files include JSON schema (in a text file) against which the application descriptor is validated, when a new application is created. Also, some examples of application descriptors can be found in the `ApplicationDescriptorExamples` folder.

**Helpers** This folder contains helpers that are useful in different scenarios and because of this, we will elaborate on each of them individually.

- `AuthorizationHelper.cs` – this helper contains methods for authorization that can be used by client applications to verify that a user has access to the data.
- `Logger.cs` – this helper contains loggers. In the future loggers to files could be added into this file.
- `MessageHelper.cs` – this helper contains methods for creating frequently used messages. At the moment only server error message is present in this helper.
- `SharedAppInitHelper.cs` – this helper is used for performing application descriptor validation defined in subsection 3.7.1.
- `SharedValidationHelper.cs` – this helper contains methods for various validations, be it a validation of rights or input data of a dataset.

- `TokenHelper.cs` – this helper is used for accessing data contained in the JWT token received from the Server application.

**Models** This folder contains the mapping of database tables to the C# classes. Based on these models the database can be created. If the database needs to be altered, changes in these models and a migration of the database needs to be made.

**Services** This folder contains classes with methods for accessing the API endpoints of the Server application. The services inherit from `BaseService` class that defines the base address of the API endpoints. In case of changes on the API, these services need to be changed as well accordingly.

**StaticFiles** This folder contains classes with constants and plain texts. In the future contents of this classes should be moved into plain text files and configuration files.

**Structures** This folder contains structures that are useful for both the Server and the Web client application or other client applications. These structures include a structure of JWT token, structure for login credentials, messages and a structure used when a password is changed.

Content of all of the folder within the `SharedLibrary` project was introduced in the previous enumeration. For readers interested in more details about individual classes we recommend looking through the `SharedLibrary` project itself since documentation comments might be helpful in understanding the full details of each class.

## 4.4 Issues

During development we have encountered some issues worth mentioning.

### 4.4.1 HTTPS support

As mentioned in section 3.1 we would like to use an HTTPS connection when a client application connects to the MetaRMS API. During development, we have encountered some problems with achieving this and we will describe them in the following subsection.

To allow HTTPS connection on localhost, we had to create a self-signed certificate. Because our development platform was macOS, we have generated the certificate on this platform. For this, we have found the

article [Car17] helpful. This article also mentions how to generate self-signed certificates for other platforms.

To enable HTTPS connection it is important to set variable `UseHttps` in `Constants` class in the `SharedLibrary` project to true. Also path to the certificate must be assigned to the `HttpsCertificatePath` variable and password to the certificate to the `HttpsCertificatePassword` variable.

The HTTPS communication was tested from Postman (an API testing tool—[www.getpostman.com](https://www.getpostman.com)). In figure 4.7 we can see a successful login to the secured page.

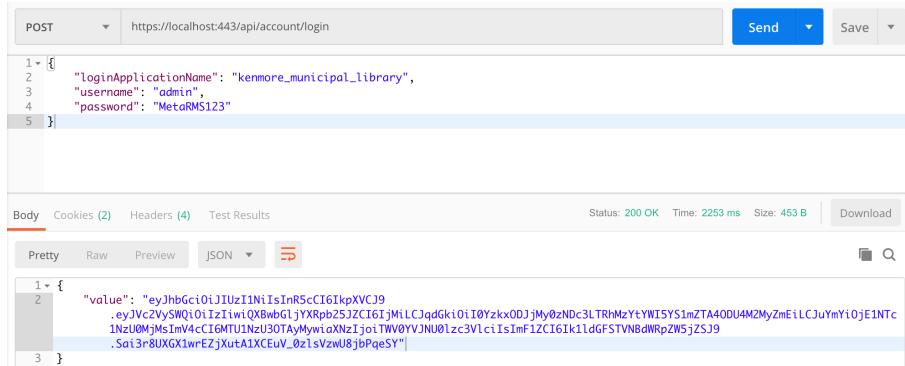


Figure 4.7: Testing HTTPS secured API with Postman.

When testing the HTTPS communication from the Web client application we have encountered a `System.PlatformNotSupportedException` exception caused by an incorrect libcurl version. As we have understood from discussion [Git] this problem should be solved after upgrading to the higher version of .NET Core, which is one of our future goals as mentioned in section 6.1.

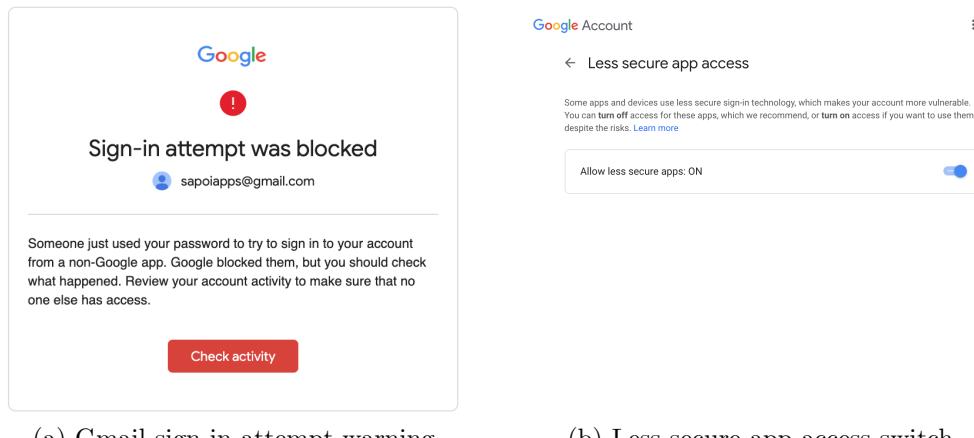
Because of the problems with client application we have decided to turn HTTPS connection off by default. This feature is for now just experimental and can be turned on only with the debug configuration.

Unfortunately, the free version of the hosting [www.aspify.com](https://www.aspify.com), that we are using, does not have SSL support. Because of this the HTTPS communication was not tested in the production environment. This is one of the goals in the Future work mentioned in section 6.1.

#### 4.4.2 Sending emails

For automatic sending of emails we have used a Gmail account. To be able to send emails from the production server, we had to allow access to less secure applications. This can be done in the settings of the Google account. Go to Google Account → Security → Less secure app access → Turn on access

(not recommended) → and here turn the switch to blue. Figure 4.8 shows (a) error message received when sending the email with Less secure app access turned off and (b) how the Less secure app access settings should look so that the emails can be sent.



(a) Gmail sign-in attempt warning.

(b) Less secure app access switch.

Figure 4.8: Gmail settings to allow sending emails from MetaRMS.

#### 4.4.3 Cultural info

In the production environment we have received `System.FormatException` when converting strings to floats. This was caused by a different culture of the production server. Setting the `CultureInfo` to `CultureInfo.InvariantCulture` when parsing has solved this problem.

#### 4.4.4 Login length on production

After deploying and testing the deployed version of MetaRMS on the production server, we have found that after login the user is logged out after 1-3 minutes. We have not found what this time depends on nor have we found what is this log out caused by. Changing of the expiry time of the JWT issued on the server also did not produce any changes.

Because this problem is not present when running MetaRMS locally, we suppose, that this problem is caused by the hosting. We have discussed the problem with hosting helpdesk. Their advice was to check the Application Pool restarts in the Log Manager, but those restart times did not match the logout times. Unfortunately, we have not managed to solve this issue.



# 5. User guide

This chapter will contain guides for different users of MetaRMS. The following enumeration divides these users into groups:

1. Developers of the client applications interacting with the open JSON API. With this API, anyone can create a client application for an arbitrary platform and connect it to the MetaRMS. This will be elaborated in more detail in section 5.1 of this chapter.
2. System administrators who can edit the source code of MetaRMS server and web client application. These administrators need to be able to run and debug the application locally and deploy the final version to the production server. Section 5.2 will contain a guide for this group of users.
3. Application administrators who create an application based on an application descriptor and then administrate it for their clients, set user accounts and authorization levels. These administrators will create the application on an already running instance of MetaRMS. This will be elaborated in more detail in section 5.3 of this chapter.
4. Application users, who log in to an already generated and prepared application and manipulate with the data within their authorization level. This group of end-users will be elaborated in more detail in section 5.4 of this chapter.

It is expected, that these groups are not disjunct and that for example, an application administrator can at the same time be an application user (especially for personal-use applications).

## 5.1 Client application developer's guide

This section targets developers, who want to create a client application for MetaRMS. Their main concern will be the JSON web API and the Shared library mentioned in subsection 4.3.2.

To create a client application it is important to understand the processes within MetaRMS. These processes are in detail described in the section 4.2. The Swagger documentation of the API can be found online on [sapoi.aspifyhost.com/api/swagger](http://sapoi.aspifyhost.com/api/swagger). For local instance of MetaRMS, the documentation is on <http://localhost/api/swagger> or

on `https://localhost/api/swagger` if HTTPS is enabled. How to run MetaRMS locally is described in the following section 5.2.

It is recommended to use the SharedLibrary when developing client applications, since it contains functions for validations based on the application descriptor, functions for user authorization, etc. It is a .NET Standard 2.0 library written in C# and it is in detail described in subsection 4.3.2.

## 5.2 System administrator's guide

As a system administrator, we consider a person who mainly deploys an instance of MetaRMS on their server. This process may include performing changes in the MetaRMS code, testing and debugging. In this section, local testing of MetaRMS will be explained in Step 1 and the actual deployment to the server in Step 2. In Step 3 the database will be created and added to the server.

In case that the administrators want to first try the generated applications, the option is on the already running instance of MetaRMS located on `sapoi.aspifyhost.com`. The applications mentioned in this section in Step 1 in substep 4 are with the same login credentials present also on this instance.

### Step 1 – Local testing

To test the functionality of MetaRMS first locally, it is important to support .Net Core 2.0. It takes the following steps to run MetaRMS on a local system.

1. Download the latest source files from `github.com/sapoi/MetaRMS`. It is also possible to use MetaRMS source codes attached to this thesis.
2. Set your own values in the `StaticFiles/Constants.cs` file in the `SharedLibrary` project. The values that will need to be changed are in the `Application initialization email settings` region. When experimenting with the HTTPS connection, variables in the `Security settings` region might also be changed.
3. After that the application is ready to be executed and debugged.

**macOS** The project already contains `.vscode` folder with the configuration for Visual Studio Code. When the project is opened in the Visual Studio Code for the first time, all the dependencies

must be restored, as shown in the figure 5.1. Then the project can be debugged or ran.

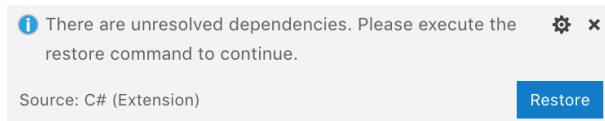


Figure 5.1: Restoring dependencies in Visual Studio Code.

**Windows** In Visual Studio select **Core** as the startup project and then select **Core** again. This selection is depicted in the figure 5.2.

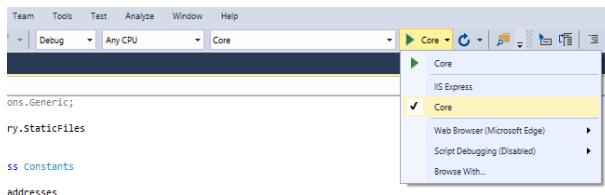


Figure 5.2: Settings in Visual Studio.

4. The attached database already contains 6 examples of generated applications. These applications corresponded to our representative scenarios mentioned in the Representative scenarios subsubsection of section 1.1. Each of these applications contains an account to which the administrator can log in and explore the application. The login application names are respectively “ats”, “inventory”, “kenmore\_municipal\_library”, “package\_delivery”, “sports\_tracker” and “todo\_list”. For these examples, the username is always “admin” and the password is “MetaRMS123”. On top of that the “kenmore\_municipal\_library” contains additional accounts with different rights sets. The password to these accounts is always “MetaRMS123” and the usernames are “Joseph”, “Anna”, “Elisabeth”, “Peter”, “Mike” and “Julia”. Details about the individual descriptors will be elaborated in section 5.3.

## Step 2 – Deployment to the server

If we want to deploy MetaRMS to our own server, we need an ASP.NET Core server with ASP.NET Core 2.0 and Entity Framework Core 2.0 support. For this we used a free hosting server [aspify.com](http://aspify.com) selected in section 3.9.

This step builds on the Step 1 and starts with a working version of MetaRMS, that is able to run locally. This tutorial expects, that the administrator has at least minor experience with terminal.

1. Set location of the server into the `ReleaseServerBaseAddress` variable in in the `StaticFiles/Constants.cs` file in the `SharedLibrary` project.

2. Go to the `Core` folder in terminal

```
cd Core/
```

3. Deploy MetaRMS with Release build configuration

```
dotnet publish -c Release
```

4. Copy all the contents of the folder `Core/bin/Release/netcoreapp2.0/publish/` to the `www` folder of the hosting server

### Step 3 – Database creation

The source codes already contain a prepared database in the `Core/database.db` file. This database contains prepared applications as mentioned at the end of the Step 1 in this guide.

If we want to create our own empty database the following steps should be followed. Otherwise, the already prepared database can be used as well so only the substep 5 from the following list needs to be done.

1. Delete the `Core/database.db` file and `Core/Migrations` folder

2. Go to the `Core` folder in the terminal

```
cd Core/
```

3. Scaffold a migration to create the initial set of tables for the model

```
dotnet ef migrations add <name of the migration>
```

4. Create a database and apply the new migration to it

```
dotnet ef database update
```

5. Copy the new `Core/database.db` file to the `www` folder of the hosting server

## **Step 4 – Test the connection**

This is the last step of the system administrator’s tutorial. At the moment MetaRMS should be running on the server.

## **5.3 Application administrator’s guide**

This section contains a guide for the application administrators. By application administrator, we mean a user who writes an application descriptor and generates the application via already running instance of MetaRMS. This means that the administrator must understand the format of the application descriptor and the process of creation of new application users and setup of their rights.

### **Step 1 – Create an application descriptor**

Chapter 2 describes all the necessary parts of the application descriptor and the reasons for their inclusion in the final solution. Section 3.7 then contains a selection of the format of the application descriptor together with the basic structure. Finally, the result structure of the application descriptor can be found in subsection 3.7.3.

The following subsubsection contains advice for writing an application descriptor. Inspiration can be found in the already existing application descriptors and in the Examples subsubsection at the end of this guide. Also, it is recommended to read Best practices mentioned later in this subsubsection, before writing an application descriptor.

As mentioned in the previous paragraph, complete examples of application descriptors can be found as a part of the `SharedLibrary` project in the `Files/ApplicationDescriptorExamples` folder and this folder also contains a basic application descriptor structure with comments—file `application_descriptor_basic_structure_explanation.json`. JSON schema against which the descriptor is validated, can be found in the `SharedLibrary` project in the `Files` folder, in the file `jsonschema.json`. The following enumeration contains a brief description of each of the application descriptor from examples.

**`ats.json`** This application descriptor contains an example of how to create a chain of records. The Interviews dataset can contain references to Previous and Following interviews.

**`inventory.json`**

This application descriptor contains examples of enumerations, such

as categories, employees positions, and currencies.

`kenmore_municipal_library.json` This descriptor contains an application, where different levels of rights are necessary. The reason is, that the Payroll dataset should not be visible to every employee of the library, but only to accountants. Creation of the rights sets and the user accounts is elaborated in Steps 3 and 4 of this guide.

`package_delivery_cz.json` Even though the language settings in descriptor has to be set to English, which results in English button descriptions and messages, this example shows some parts of the descriptor written in the Czech language. These values are names of datasets, their attributes, and descriptions. The language different from English was selected on purpose to show, that these values can be in an arbitrary language.

`sports_tracker.json` This simple application descriptor creates a template for general sports trackers and it shows support of emojis in the descriptions.

`todo_list.json` This easily extendable application descriptor can be used as an inspiration for simple ToDo applications. Also it uses all types of onDeleteActions mentioned in bullet 7 of subsection 2.3.1.

Best practices for writing the application descriptor:

- Write the most frequently used datasets first because the datasets in the application menu will be displayed in the order in which they are in the application descriptor.
- Write attributes with the important or distinctive information first in a dataset. The first 3 attributes are displayed when a record of the dataset is referenced and the records in one dataset should be distinguishable by the values of the first 3 attributes. More information about this can be found in section 3.3.

## Step 2 – Create the application

In this step, the application descriptor is submitted to the MetaRMS server and validated. If the descriptor is correct the application is created and the administrator can log in.

1. Upload the application descriptor and email address to MetaRMS. The screenshots in the figure 5.3 come from the provided web client. Figure

(a) shows the initialization page on a desktop device and figure (b) on a mobile device.

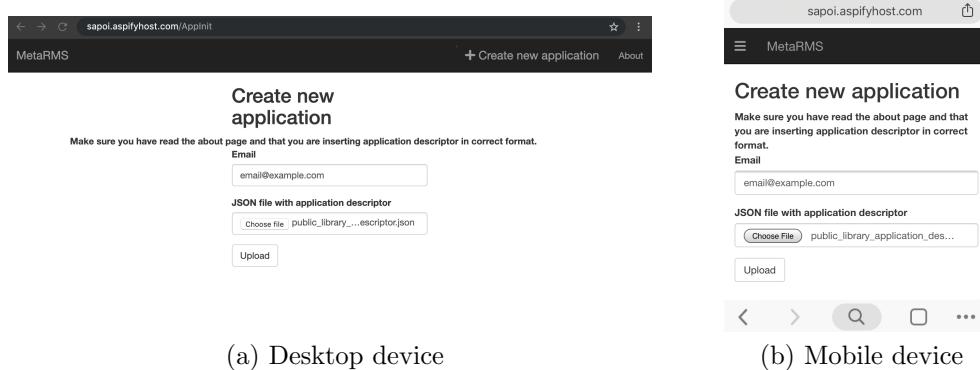


Figure 5.3: Application initialization page on various devices.

2. After submitting, the application descriptor is validated on the server side. If there are any errors found, they are displayed on the application initialization page and the application is not created. Example of the page with errors is in the figure 5.4.

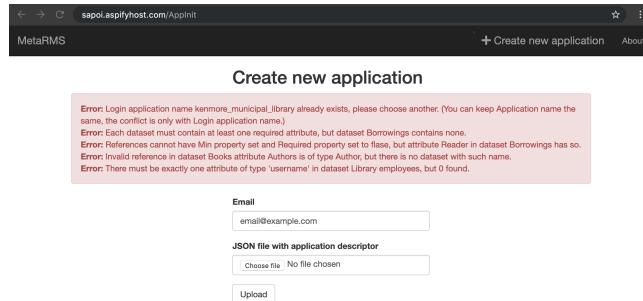


Figure 5.4: Example of displayed errors after submitting invalid application descriptor.

3. After a valid application descriptor is submitted, wait for an email with login credentials. Example of such email is in figure 5.5.

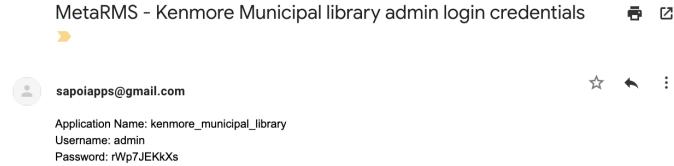


Figure 5.5: Example of email with login credentials.

4. Log into the newly generated application with received credentials. The login page displayed after application initialization is in figure 5.6.

Figure 5.6: Login page after application initialization.

### Step 3 – Setting user rights

Section 3.5 contains a detail description of the system of user rights. When creating a set of rights, it is important to understand the meaning of each of the rights levels.

**None** No rights at all

**R** Only right to read the data.

**CR** Right to read existing and create new data.

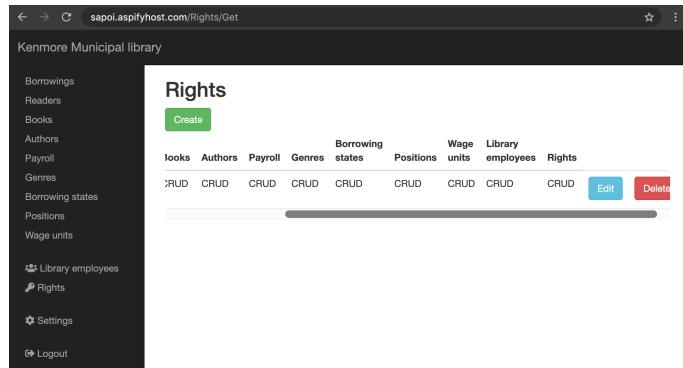
**CRU** Right to read and update existing and create new data.

**CRUD** Full rights—read, create, update and delete the data.

New rights set can be created on the `/Rights/Create` address or after clicking the Create button (the green one) on the Rights page—Rights page is in figure (a) of figure 5.7. On the create page—displayed in the figure (b) of figure 5.7—the name of the new rights set and the level of the rights for each dataset must be filled. An important thing to keep in mind, mentioned in the Rights validation subsubsection of section 3.5, is that datasets referenced

in a dataset with a read right must also have the read right. The valid rights set is saved by clicking the Save button.

The rights set can be modified by clicking the Edit button (the blue one) or deleted by clicking the Delete button (the red one).



(a) Page with all rights sets.

This screenshot shows a "Create" page for a new rights set. The URL is `sapoi.aspiflyhost.com/Rights/Create`. The main title is "Kenmore Municipal library". The sidebar is identical to the previous screenshot. The main content area is titled "Rights" and contains a "Name" input field. Below it are sections for "Borrowings", "Readers", "Books", "Authors", "Payroll", "Genres", and "Borrowing states", each with a dropdown menu showing "None".

(b) Page for creating a new rights set.

Figure 5.7: Pages for creating new rights sets in the application.

#### Step 4 – Setting user accounts

When the application was generated at the Step 2 of this guide, only the administrator account was created and the login credentials were sent to the email address provided. To allow other users access this application we need to create their accounts first.

Creation of a new user account is done on the `/User/Create` page displayed in the figure (b) of the figure 5.8. This page can be accessed by clicking the Create button (the green one) on the Users page displayed

in the figure (a) of the figure 5.8. The structure of this page depends on the structure of the application descriptor. Every user must have a unique username and must have a rights set assigned. The other fields and their valid values are application-specific. If all the filled data are correct, a new user is created by clicking the Save button. The password of the new user is set to “MetaRMS123” and it is strongly advised to change it as soon as possible.

The user can be modified by clicking to the Edit button (the blue one) or deleted by clicking the Delete button (the red one). The Reset password button (the orange one) resets user’s password to the default “MetaRMS123” value.

This screenshot shows a web page titled "Library employees" from the URL [sapoi.aspifyhost.com/User/Get](http://sapoi.aspifyhost.com/User/Get). The page displays a table with four columns: Username, Position, Boss, and Rights. A single row is shown for the user "admin". Below the table are three buttons: "Edit" (blue), "Reset password" (orange), and "Delete" (red). The left sidebar contains links for Borrowings, Readers, Books, Authors, Payroll, Genres, Borrowing states, Positions, Wage units, Library employees, Rights, Settings, and Logout. The title bar says "Kenmore Municipal library".

(a) Page with all users.

This screenshot shows a web page titled "Library employees" from the URL [sapoi.aspifyhost.com/User/Create](http://sapoi.aspifyhost.com/User/Create). The page has a form with four input fields: "Username" (Login username), "Position" (Users position at the library), "Boss" (Employee's boss and person responsible for checking responsibilities), and "Rights". Each field has a placeholder text and a dropdown arrow. Below the form is a "Save" button. The left sidebar contains the same links as in figure (a). The title bar says "Kenmore Municipal library".

(b) Page for creating a new user.

Figure 5.8: Pages for creating new users in the application.

## Examples

The following subsubsection contains some examples from the application descriptor. Each paragraph describes one example followed with the code

representation.

The example of the main structure is shown in the following code. The name of the application is “My application” and the application name used when logging into the application is “my\_application\_login\_name”. The default language is set to the English language since it is the only currently supported language. The users dataset in system datasets and user-defined datasets will be elaborated in the following examples.

```
1 {
2     "ApplicationName": "My application",
3     "LoginApplicationName": "my_application_login_name",
4     "DefaultLanguage": "en",
5     "SystemDatasets": {
6         "UsersDatasetDescriptor": {
7             ...
8         }
9     }
10 }
11 }
```

The system users dataset example shows dataset with application users named “Application users”. This name can be used when referencing this dataset. This dataset contains password attribute and other attributes, described in more detail in the following examples.

```
1 "UsersDatasetDescriptor": {
2     "Name": "Application users",
3     "Description": "This dataset contains all users of my application.",
4     "PasswordAttribute": [...],
5     "Attributes": [...]
6 }
```

User-defined datasets are represented as a list of datasets. This example shows the structure of one such dataset with the name “Books”. Each dataset must contain some attributes, which will be elaborated in the following examples.

```
1 {
2     "Name": "Books",
3     "Description": "This dataset contains books.",
4     "Attributes": [...]
5 }
```

This example shows how to create a Basic type attribute. This attribute is named “Book name” and has a description “This attribute contains the name of the book.”, which will be displayed next to the attribute to help users understand the content. Type of the attribute is “string”, which is one of the Basic types mentioned in the bullet 3 of subsection 2.3.1. This attribute is required and the content of it must be between 5 and 50 characters long. For more details about the settings of minimum and maximum, read the bullet 6 of subsection 2.3.1.

```
1 {
2     "Name": "Book name",
3     "Description": "This attribute contains name of the book.",
4     "Type": "string",
5     "Required": true,
6     "Min": 5,
```

```

7     "Max":50
8 }
```

The following is an example of a Reference type attribute. For this, we expect that the descriptor contains dataset named “Authors”. It is not important whether the “Authors” dataset is defined before or after it is used as a type. The OnDeleteAction of this attribute is set to “protect”, which means that an Author cannot be deleted if it is referenced within any of the Book records. More information about OnDeleteActions can be found in the bullet 7 of a subsection 2.3.1.

```

1 {
2   "Name": "Authors of the book",
3   "Description": "This attribute contains authors of the book.",
4   "Type": "Authors",
5   "Required": true,
6   "Min": 1,
7   "OnDeleteAction": "protect"
8 }
```

One of the special attributes is the Username attribute described in section 2.3. This attribute must be located only in the System users dataset within the list of attributes. The type of the Username attribute must be “username” and it must be required and unique. The setting of a minimal and a maximal length of the username is optional.

```

1 {
2   "Name": "User's username",
3   "Description": "This attribute contains a username. The username us used
4   for login.",
5   "Type": "username",
6   "Required": true,
7   "Unique": true,
8   "Min": 3,
9   "Max": 20
9 }
```

Another special attribute is the Password attribute defined in section 2.3. This attribute is within the System users dataset, but it is not in the list with other attributes. The type of Password attribute must be “password” and it must be required. When the safer property of the password attribute is set to true, special requirements are enforced as mentioned in the Password policy subsubsection of section 3.4.

```

1 "PasswordAttribute": {
2   "Name": "User's password",
3   "Description": "User's password. It is required that the password
4   contains lower- and upper-case letter and number.",
5   "Type": "password",
6   "Required": true,
7   "Safer": true,
8   "Min": 10,
9   "Max": 20
9 }
```

## 5.4 End-user's guide

This guide targets end-users of the applications generated by MetaRMS. These users do not need to understand the structure of the application descriptor. They can simply log into the application and manipulate the data, based on their autohrization level.

Since every application will have a different structure, this guide will use a library application from representative scenario 2a to show the general functionality of the generated applications.

### Login

The login form is located on the main page of MetaRMS. This address should be provided by the application administrator. The figure 5.9 shows login on a desktop (a) and a mobile (b) device. It is necessary to fill the application name, username, and password. The application name and username should be provided by the application administrator. For the first log in, the password is “MetaRMS123” and it is important to change this password as soon as possible. The process of changing the password will be described later in this guide.

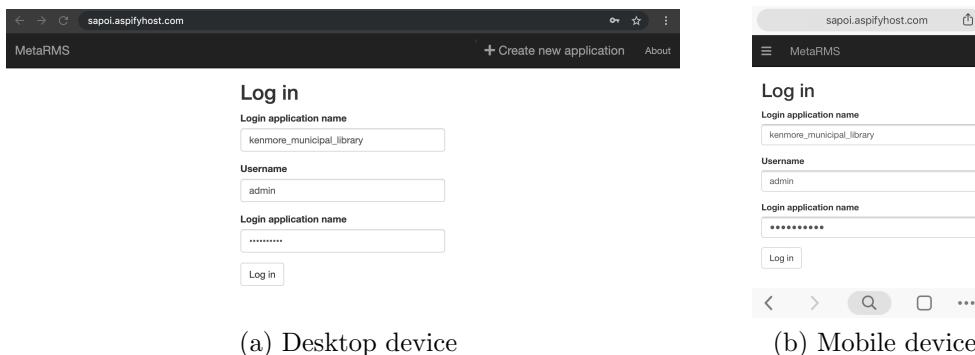


Figure 5.9: Login page on various devices.

### User interface

The figure 5.10 shows the user interface on a page with a dataset content, on (a) a desktop device and (b) a mobile device. On the desktop devices, there is a dark grey panel on the left with all the read-accessible datasets to the logged user. The panel also contains Settings and a Logout button. On the mobile devices, this panel is hidden under the three-lined button (also called a “hamburger” button). The main area of the application is for displaying

the data. Based on the rights for the dataset, the Create (green), Edit (blue) and Delete (red) buttons may be displayed.

The figure shows two screenshots of a web application interface for managing library readers. Both screens have a header bar with the URL 'sapoi.asphyhost.com/Data/Get/Readers' and the library name 'Kenmore Municipal library'. On the left, under 'Readers', there is a table with five rows of data:

Full name	Library ID	Birth date	Library ID valid until	Email address	Telephone	Edit	Delete
Leslie Parry	001	1967-07-18	2019-12-31	leslie.parry@kenmore.uk	12 5648 3368 83	<a href="#">Edit</a>	<a href="#">Delete</a>
Erin Hamilton	002	2000-05-14	2019-08-31	erin.hamilton@kenmore.uk	075 (437) 36 744	<a href="#">Edit</a>	<a href="#">Delete</a>
Reed Cooper	003	1993-01-22	2019-09-30	reed.cooper@kenmore.uk	105 553 827 12	<a href="#">Edit</a>	<a href="#">Delete</a>
Phoenix Grant	004	1987-05-01	2020-01-31	phoenix.grant@kenmore.uk	955 6990 53 45	<a href="#">Edit</a>	<a href="#">Delete</a>

A green 'Create' button is visible above the table. The right screenshot shows a similar view on a mobile device, with the same table and 'Create' button, but with smaller text and a different layout.

Figure 5.10: Page with content of a dataset on various devices.

## Creating new data

A new record in a dataset can be created after clicking on the Create button (the green one) on the page with the dataset—this page is displayed in figure 5.10. After that, a page with empty fields based on the dataset structure is displayed. This create page is shown on (a) a desktop device and (b) a mobile device in figure 5.11. Based on the type of an attribute the field can accept a string, number, date and time, etc. In the example figure, the calendar input is shown. After the data are filled they are saved by clicking the Save button.

**(a) Desktop device**

**(b) Mobile device**

Figure 5.11: Page for creating a new record in a dataset on various devices with the calendar input displayed.

## Updating existing data

Already existing record in a dataset can be modified after clicking on the Edit button (the blue one) on the page with the dataset—this page is displayed in figure 5.10. After that, a page with filled editable fields of the record is opened. This edit page is displayed on (a) a desktop device and (b) a mobile device in figure 5.12. Changes can be saved by clicking the Save button.

**(a) Desktop device**

**(b) Mobile device**

Figure 5.12: Page for editting an already existing record in a dataset on various devices.

## Deleting data

Deletion of data is performed by clicking the Delete button (the red one) as displayed in the figure 5.10.

## Settings

In the settings section of the application, displayed in figure 5.13, the logged user can change their own password. As mentioned in the Login subsubsection it is important to change the password after the first login. To change the password fill the old one and twice the new one.

(a) Desktop device

(b) Mobile device

Figure 5.13: Page for changing password of the logged user on various devices.

# 6. Conclusion

The goal of this thesis was to develop an own application generating software, that will satisfy all groups from the Representative scenarios subsubsection of section 1.1 and fulfill the requirements from section 1.3. In this chapter, we will revisit each of the requirements again and review if the requirement was achieved.

**R1 Application generating software is required to be able to take a definition of various structures and thus create different applications.**

As mentioned in subsection 4.3.2 and section 5.3 the folder `SharedLibrary/Files/ApplicationDescriptorExamples` contains application descriptors for all our representative scenarios mentioned in the Representative scenarios subsubsection of section 1.1. These scenarios were selected, because they represent various structures for both individuals and companies of various sizes. Since we were able to create all of the descriptors, we can consider this requirement as met.

**R2 Application generating software is required to be multiplatform.**

By selecting .NET Core as platform in section 3.2 we have ensured that MetaRMS can run on various platforms. We have tried to run MetaRMS on macOS Mojave 10.14.4 (where the system was developed), and Windows 10 Version 1809.

**R3 Applications generated by the application generating software must be easy to use for end users.**

The decision about this requirement is based on personal preferences of each user, since we have not made any public research nor have we questioned any end-users. However, we have made the web client application with standard web elements (buttons, tables, descriptions) to make it look familiar for the end-users.

**R4 Application generating software should provide an easy way to create a new application.**

A new application is generated based on the application descriptor provided. The whole structure of the descriptor was elaborated in

chapter 2. For huge systems, this structure might get a bit complicated, but for regular systems, we find it reasonable.

**R5 Time delta between creating a new application and its usage by the end-users should be minimized for applications created by the application generating software.**

As mentioned in section 5.3 after a valid application descriptor is submitted, an email with the administrator's login credentials is sent and the application is ready to use. To prepare it for end-users the administrator must create user accounts and appropriate rights sets as described in the same section.

**R6 Applications generated by the application generating software must contain authentication.**

Authentication was elaborated in section 3.4 and in this way it was also implemented.

**R7 Applications generated by the application generating software must contain authorization.**

Authorization was elaborated in section 3.5 and in this way it was also implemented.

**R8 Applications generated by the application generating software should have multilingual support.**

As decided in section 2.1 due to limited resources we support only the English language. Still, MetaRMS was implemented with this requirement in mind so the software is ready for this feature to be implemented in the future.

**R9 Application generating software should have an API so anyone can use it as an endpoint.**

As described in section 5.1 MetaRMS contains public JSON web API with Swagger documentation. This API can be used by other developers to create client applications.

**R10 Application generating software should have an easily extendable source code.**

We believe, that by implementing our design analyzed in chapter 3, the code is reasonably structured and well designed.

After summarizing all the requirements, we consider the the golas of this thesis to be fulfilled. Even though we have found possible improvements, that could be implemented in the future to make MetaRMS better. These ideas will be introduced in the following section 6.1.

## 6.1 Future work

- *Extend time the user is signed on the production server* – As mentioned in subsection 4.4.4 we have encountered a problem when a user logs in on the production server. In the future, this will be the first issue to solve.
- *HTTPS support on production server* – Solving this issue would mean migrate MetaRMS to a paid server and get an SSL certificate. After that users would not need to worry about the transfer of their data.
- *Data filtering and searching* – With the increasing amount of stored data in individual applications users will need to search and filter their data. To satisfy them both of these two features will need to be implemented.
- *Add pages with data and user detail* – At the moment details are displayed on the edit page, thus inaccessible to users without sufficient rights. In the future, separate pages for data and user details should be created.
- *Visual creator for application descriptors* – As mentioned in section 1.4 some of the low-code and no-code platforms contained visual creator of the application. To make the process of creating a new application more user-friendly this will be a great component.
- *Improve the logout process on the server* – As mentioned in the Logout subsubsection of subsection 4.2.2 the process of logout on the server side was simplified. In a further development, this process should be implemented in the way described in subsection 4.2.2.
- *Migrate to higher version of .NET Core* – MetaRMS runs on .NET Core version 2.0, which is no longer supported. Because of this, it should be migrated to a newer version. As recommended at the date of writing the thesis on website [Dot] .NET Core version 2.2 would be preferred.

- *Finish the support for different languages* As already mentioned in this chapter the support for multiple languages was not implemented due to limited resources. In the future, this feature should be finished.
- *Allow later modifications of application descriptor* – In the future the administrators might want to change some parts of the application descriptor. It would be important to decide what parts of the descriptor could be changed and what would not.

# Bibliography

- [Cin17] Jindřich Cincibuch. “Qubit - systém pro správu a tvorbu překladů”. Bachelor Thesis. Charles University, 2017.
- [Dot] *Download .NET Core*. URL: <https://dotnet.microsoft.com/download/dotnet-core>.
- [Eug] *2018 reform of EU data protection rules*. 2018. URL: [https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules\\_en](https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en).
- [G2l] *Best Low-Code Development Platforms Software*. 2019. URL: <https://www.g2.com/categories/low-code-development-platforms>.
- [G2n] *Best No-Code Development Platforms Software*. 2019. URL: <https://www.g2.com/categories/no-code-development-platforms>.
- [Git] *The handler does not support client authentication certificates with this combination of libcurl (7.54.0) and its SSL backend (LibreSSL/2.0.20)*. 2018. URL: <https://github.com/dotnet/corefx/issues/27000>.
- [Nal18] Martin Nally. *REST vs. RPC: what problems are you trying to solve with your APIs?* 2018. URL: <https://cloud.google.com/blog/products/application-development/rest-vs-rpc-what-problems-are-you-trying-to-solve-with-yourapis>.
- [Sal] *Salted Password Hashing - Doing it Right*. 2018. URL: <https://crackstation.net/hashing-security.htm#salt>.
- [W3r] *Usage of server-side programming languages for website*. 2019. URL: [https://w3techs.com/technologies/overview/programming\\_language/all](https://w3techs.com/technologies/overview/programming_language/all).
- [Wika] *Cross-site request forgery — Wikipedia, The Free Encyclopedia*. 2019. URL: [https://en.wikipedia.org/wiki/Cross-site\\_request\\_forgery](https://en.wikipedia.org/wiki/Cross-site_request_forgery).
- [Wikb] *Cross-site scripting — Wikipedia, The Free Encyclopedia*. 2019. URL: [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting).



22dateInterval%22%3A%22Monthly%22%2C%22dateStart%22%  
3A%222018-01%22%2C%22dateEnd%22%3A%222018-12%22%2C%  
22segments%22%3A%22-1000%22%7D.

- [Pra14] Prasad Kharkar. *JPA Single Table Inheritance Example*. 2014. URL: <http://www.thejavageek.com/2014/05/14/jpa-single-table-inheritance-example/>.
- [Pro16] Prosper Otemuyiwa. *JSON Web Tokens vs. Session Cookies: In Practice*. 2016. URL: <https://ponyfoo.com/articles/json-web-tokens-vs-session-cookies>.
- [Ric19] Rick Anderson, Ryan Nowak. *Introduction to Razor Pages in ASP.NET Core*. 2019. URL: <https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-2.0&tabs=visual-studio>.
- [Rya18] Ryan. *Code-First vs Model-First vs Database-First: Pros and Cons*. 2018. URL: <https://www.ryadel.com/en/code-first-model-first-database-first-vs-comparison-orm-asp-net-core-entity-framework-ef-data/>.
- [Wil16] William R. Stanek. *Best Practices for Enforcing Password Policies*. 2016. URL: [https://docs.microsoft.com/en-us/previous-versions/technet-magazine/ff741764\(v=msdn.10\)](https://docs.microsoft.com/en-us/previous-versions/technet-magazine/ff741764(v=msdn.10)).

The validity of all the links was checked on 13-May-2019.



# Attachments

## Attachment A — the Enclosed CD

Contents of the attached CD

```
src - folder with source codes
  .vscode - folder with the configuration for Visual
    Studio Code
  Core - the Core project described in subsection 4.3.1
  SharedLibrary - the SharedLibrary project described in
    subsection 4.3.2
  MetaRMS.sln - information about the MetaRMS solution
  thesis.pdf - file containing this thesis
  README.txt - file describing the contents of this CD
  LICENSE.txt - file containing licensing information
```

