Project Report

SWTlib - Library Rental and Management System

SWT-PR[1|2]-[B|M]
Summer Semester 2019

Group B

Simon Alexander Nützel
Student Number: 1787792
Degree Course/Semester: SoSySc/5

Sergej Gelver
Student Number: 1822695
Degree Course/Semester: Angewandte Informatik/5

Daniel Rooney
Student Number: 1979717
Degree Course/Semester: SoSySc/6(ERASMUS)

Zdenek Scherrer
Student Number: 1706252
Degree Course/Semester: Angewandte Informatik/6

Lewis Jermy
Student Number: Lewis you need to fill this in you fanny baw
Degree Course/Semester: SoSySc/6(ERASMUS)

Supervisor: Professor G.Lüttgen
Version: 22.07.2019

Abstract

Since the chair members of the SWT Chair needed a better solution for borrowing and managing the books owned by the chair, we have developed a web application in order to provide a convenient, online library renting system called SWTlib. To meet all functional requirements such as "Display Catalogues", "Search for Specific Terms" or "Borrow Books", we first needed to get our Scope of Work and to also write user stories. After some discussion we decided to use ASP.NET Core MVC as our main architecture combined with Entity Framework Core as a communication layer between the application and the database as well as MySQL as our database. After a feedback meeting with AraCom we decided to not use a clientside framework such as AngularJS or ReactJS, but rather stick to plain ASP.NET Core paired with the already included Bootstrap 4 and a small amount of jQuery. Within the implementation process we ran into several issues caused by our lack of experience regarding the chosen architecture as well as programming in general. We had some issues to get our team working and assign appropriate amount of work to each team member. With our evaluation we could determine some major bugs, such as unhandled exceptions which crashed the page. We were able to achieve most of the requested requirements and functionalities. There is definetly a lot of space for improvement and tweaking if the software should be used. Nevertheless we managed to implement a working solution with almost zero programming experience within a architecture we were unfamiliar with and had to manage ourselves within an agile development.

Contents

Contents

Glossary

Assumption: A thing that is accepted as true or as certain to happen, without proof.

Constraint: Global requirement that represents a restriction.

Requirement: A capability that shall be possessed by a system to satisfy a specification.

SWTlib - Library Rental and Management System

1   Problem Description

Approximate weighing on mark: 10%
Approximate percentage of expected report length: 7.5%

Provide an overview of your project by describing the problem, including the project's context and goal, and your employed methodology.

1.1   Context and Goal

Detail the context and goal of your project. You may describe the goal by stating the project's purpose, its advantage to, e.g., end users or the SWT Lehrstuhl, and how the achievement of the goal can be measured.

Goal of SWTlib

SWTlib has been developed in order to simplify the Tasks involved when Managing a Library, so that those Tasks can be tackled and goals can be achieved faster, allowing the Chair members to dedicate more time to research and teaching. These Tasks include, but are not limited to, Display the Catalogue, rent, add, edit, wish, return and inform about available books. So far, no electronic System is in use, which caused difficulty for Students and Chair members to localize a specific book, as the available books are stored within multiple Rooms. Furthermore, the pen-and-paper rental system has proven impractical and inefficient, and there is no current way of getting an Overview of which book(s) are actually available in the library. SWTlib Tackles each of these problems by providing a platform where all of these Tasks can be dealt with.

1.2   Methodology

State and justify the approach you have chosen to address your stated problem.

Requirements

Requirement Nr.1 - Display Catalogues

A main problem of the current system is, that there is no reliable way to get an overview of which books are available in the library as well as if the specific book is rented or not. SWTlib provides a dedicated view of all the books stored in the library, as well as an indicator about the books current rental status. Adding a new book to the library updates this list, so the user is always provided with the latest information.

This Requirement is Considered done when:
- Every Available book can be seen in the Overview
- Every Displayed book shows an indicator whether the book is available or rented
- When a new Book is added/deleted/Edited, the Overview is updated

Requirement Nr.2 - Add and Classify literature items

A Key functionality of SWTlib is to add new literature to an already existing digital library, so that students/staff/chair members can have an overview of every accessible book.

This Requirement is Considered done when:
- A new book can be successfully added
- The newly added Book can be found in the Overview
- The newly added Book can be found by using the Search functionality.

Requirement Nr.3 - Search for Specific Terms

In order for every user of SWTlib to be able to find the literature they are looking for, the Platform offers a Search functionality which allows its user to search for specific Terms or Strings and get the results they are looking for.

This Requirement is Considered done when:
- A search bar is implemented
- A Search Term can be entered and only Results featuring a the Search Term are shown
- An ISBN can be used as a Search Term

Requirement Nr.4 - Share Room/Library/Metadata (ISBN/Keywords)

Due to the fact that the available books are Stored in different rooms, it is only logical to add another layer of Information. Therefor every literature can be enhanced with Metadata such as an ISBN, which can also be used as a Search term, multiple Keywords, which also can be used as Search Terms, Authors, so that students can find out more about the author and the literature he wrote, Categories, in order for staff members and students to be able to browse through other books about the same topic, Language in which the book is written in, and Rooms, so that the person who is interested in renting that book can find out in where the literature is Stored.

This Requirement is considered done when:
- Books can Store multiple Keywords
- Books can Store multiple Categories
- Books can Store multiple Authors
- Books can be assigned to a specific Room
- An ISBN can be assigned to a Specific Book
- Books can store the Language they are written in

Requirement Nr.5 -Import Previous Database

As there was an already existing Database of books available, which is out of scope to add one by one, we added a functionality to import a previously existing Database.

This Requirement is Considered done when:
- The Books from the old Database are stored corretly within SWTlib
- Books from the old Database contain the all the Metadata require to add a new book
- Books from the old Database can be found using the Search engine
- Books from the old Database can be found in the Overview

Requirement Nr.6 - Borrow Books

Students/staff/Chairmembers are supposed to be able to rent books from the library. In Order to keep the track of who has which book and for how long he is going to keep it, the Platform shall assign the rented book to the Person who rented it. The Person can then choose for how long he wants to keep the book up to a maximum of 4 weeks. If this turns out not to be enough, he has the opportunity to extend the rental period once, up to

another 4 weeks. In addition to that, the user who rented the books will see a progress bar at the right hand sight of the layout, telling him how much time he has left to return the book. If the book return date is overdue, the progress will display some form of warning, indicating that the book needs to be returned as soon as possible.

This Requirement is considered done when:
- A user can click on rent
- The rented book is assigned to the user
- The rental period can be selected
- The return date is calculated correctly
- The progress bar is displayed on the right hand side of the layout
- The extension of a rental period only works once
- Extending a book is only possible up to another 4 weeks
- Book can be returned

Requirement Nr.7 - Inform about book Details

In Order for the User to understand what a book is about, a description section has been added where a book can be described in a short Text. This is not required when adding a book, however, it helps the User to better understand what book they want to rent and if it can help them with their studies/research.

This Requirement is considered done when:
- In the book detail view, a section for "description" has been added
- Description is not a required field to add a book
- Description has no sign limit

Requirement Nr.8 - Wishlist

Not every user is going to find the specific book they want to rent/borrow. So in order for them to ask the Chair members to add new literature the "Wishlist" functionality has been added. Here they can add a Title/ISBN of a book and add this to a list. After the whole list is done, they can generate an E-mail which will contain the literature wishes of the student in the order they arranged them in, with the E-mail address auto filled.

This Requirement is considered done when:
- A User can add a book to a Wishlist
- Items within the Wishlist can be rearranged
- Clicking on a button will generate an E-mail with an auto filled E-mail address
- E-mail contains Wishlist in arranged order

Requirement Nr.9 - Show Statistics

For a Chairmember it is of high interest to find out about specific information concerning the usage of SWTlib. For example in which month is the highest Traffic on SWTlib, which book is rent the most, how long is the average rent time of a book. With this Information the Chairmember can improve the User-experience by stocking up the most favourite book, take measurements against high average rent time and many more.

This Requirement is considered done when:
- Chairmembers have access to a closed area called "Statistics"
- Several data are displayed in form of Graphs

Requirement Nr.10 - Reminders

Not always will a User be able to rent a book, as it may be unavailable in that specific time. In order to not miss the chance to rent the book as soon as possible, the user is able to set a Reminder, which will then appear on the left hand sight of the layout, displaying the title of the book he wants to be reminded of, and the date the book is expected to be returned. If the book was not brought back in time, the Reminder will display a message saying that the book is not returned yet. If the book is returned in time, the Reminder will tell you that the book is available again. Reminders can also be deleted using an "x" button right next to it.

This Requirement is considered done when:
- The "rent" button turns into a "reminder" button when a book is not available
- The Reminder is displayed on the left hand side of the layout
- The Reminder displays the correct title of the book
- The Reminder displays the correct expected return date
- The Reminder can be deleted
- If a book was not returned in time, the reminder will change to a message saying that the book is not returned yet
- If a book was returned in time, the Reminder will change to a message saying that the book is available again

2 Problem Analysis

2.1 Project Constraints & Scope

Scope

The initial scope of the project was to design and implement a standardized library rental system with essential functionality related to the process of renting and managing books, to be used by SWT Chair members . The project was not intended to exceed the original specification detailed within the project brief due to the short time period in which it was to be delivered. A breakdown of the elements that constitute the scope constraints of the project can be seen below:

Constraints

- The software must be able to run on Windows, Linux and OSX(Mac). Therefore it cannot be developed for a single operating system.

- The software must be able to run within all major browsers or browser versions post 2010. Therefore it cannot be developed with a specific browser in mind.

- The Software must be developed using ASP.NET core framework, using the MVC model so that it is easy to maintain and manage in the future.

- Project must be completed within the given deadline (30th July 2019). Therefore additional functionality should not be the main concern, only essential functionality (Dependent on time constraints and project progress).

- Users must be able to access software using their University credentials. Therefore a login is required and the application is required to be secure.

- Software must allow SWT Chair members to view and rent titles from the given catalogue, and regulate the use of the service/application by university students. Therefore a separate administration medium must be implemented.

2.2 Requirements

The requirements of the project can be separated into two dynamic entities:

- Essential functional requirements (as stated)

- Non-Functional requirements (as stated)

2.3 Functional Requirements

Below is a compiled list of all functional requirements followed by a brief summary of each, a further comprehensive explanation of each requirement is outlined in the methodology.

- **SWTlib shall be able to display a catalogue of the literature items in the SWT library**- This includes, but is not limited to; all imported titles from the existing delicious library, any new proposed titles, and titles added by request of SWT chair members.

- **Users shall be able to enter new literature items to the catalogue and classify them**- this feature is required to be simple and user friendly, and allow members to enter all required/relevant book information.

- **Users shall be able to search the catalogue for specific items by using various filter criteria**- Implementation of an advanced serach funtion with several fields.

- **SWTlib shall be able save at least the following information with each item: Room, library branch, 'ususal metadata' such as ISBN, keywords,category etc.**- This will provide users with all essential information while they use SWTlib and engage in the rental process.

- **Users shall be able to add comments / web addresses to each item** - Comment section availabe that can host links to external web pages.

- **SWTlib shall automatically add metadata to items if it is publicly available** - use of intelligent API's to compound and retrieve this data.

- **The entries of the previously used 'Delicous Library' can be imported automatically** - the ability to import XML files when using logged into the administration role/account.

- **Users shall be able to borrow books from the SWT library using SWTlib** - This eliminates the need for a paper based system, with the single exception being students who are part of the faculty (in which they will be issued a binding physical receipt.)

- **SWTlib shall give an overview of the items that are available / borrowed** - all titles displayed will display their availabilty to the user(s).

- **SWTlib shall remind users who have borrowed an item to return said item in time** - simple reminder system that notifies users of remaining rental periods.

- **SWTlib shall be able to gather and display statistical data about the usage of the library** - a statistical admin page that displays information graphically about the usage of the SWTlib software.

- **Users shall be able to create a wishlist of books they would like to see added to the library and add priorities to each item** - Users can save books to a wishlist and submit them if they are not already available within the SWTlib catalogue.

- **SWTlib shall be able to use a wishlist to automatically generate an email to the office of the Erba library secretary** - Producing an easily readable list that is automatically generated.

2.4 Non-Functional Requirements

The development team produced additional functionality that was integrated into the existing requirement list, a brief summary of each is given below:

- **Borrowing process shall be user friendly and protect the interest of all stakeholders** - Process should be simple, clear and accessible for all users, not creating any issues for both the user and stake-holders.

- **Software should be usable for at least 10 years (software license should allow SWT to continue development freely)** - to allow further development an adaption of the software in the future by development teams and admins.

- **Ensure privacy**- SWTlib must protect the privacy of it's users and their data in accordance with General Data Protection Regulations and data protection laws/practices.

- **Code should be 'high quality' and easily maintainable** - code sall be readable, well commented, easy to maintain and modify in the future.

2.5   Project Assumptions

Initial Project Assumptions

Several assumptions were raised at the blast off meeting and beginning of the project in relation to the scope. The first initial assumption being that the project would be relatively easy to build and deploy, using trusted frameworks (ASP.NET core, Entity framework core etc.) The second biggest assumption that followed was that each team member was unfamiliar with agile development practices (or held very little experience) and some of the proposed technologies.This highlighted a team wide concern that each team member would have to conduct self research/learning, with no way to gauge the rate of which each team member would be considered proficient in the proposed; and later utilized technologies. This includes the version control platform GitLab. Another articulated assumption that arose was that of the design and scope of the project, the team understood that the project had to fulfill the essential functionality outlined in the brief, breaking each element down during daily scrums into "design packages", with each team member presenting their "assumption" in terms of design of how each requirement should be implemented. This proved to change rapidly and became very fluid as the project progressed. Another assumption of the user base was made during initial meetings between team members that inventory security could not be assumed, even if the primary users are members of the SWT chair, resulting in further discussion and development of the reminder system to secure the returning of books. The final major assumption was that of team time management during each sprint and progression towards each milestone. This proved to be very fluid and change rapidly after sprint 0, once the development process was underway. With the initial assumption that we would be able to develop the product at a steady and stable pace.

Intermediate Project Assumptions

Towards the end of sprint one, the development teams' assumption base changed drastically. Several elements of the essential functionality were completed or partially completed. However several issues had arisen within this time frame. Due to communication shortcomings, individual team members ran into technical obstacles. This changed the initial assumption that we would be able to hit our deadlines for each sprint, resulting in an upheaval in productivity, due to shortage of time.The second assumption during this time frame is related to the former mentioned in this section, due to a breakdown in communication, the initial

assumption that the project would be easy to build and deploy, changed the team dynamic to a more conservative outlook in regards to time, once there was a collective realization that we would have to work closer together to achieve our goals and milestones. Thirdly, the design assumption that we should focus on styling as well as functionality, developed into a stricter focus on simplicity and essential functionality, with several non essential functions being relegated to a level of low priority.

Final Project Assumptions

The final project assumptions, towards the end of sprint two, embodied a more streamlined approach. The development team assumed that we would not hit our final milestones, at least two days overdue our set dates (team set dates). This resulted in longer, heavier pair programming sessions and more dynamic, brief discussions during progress meetings. This therefore, also resulted in clearer and more streamlined communication channels, as the major goal for all stake holders and the development team, was to deliver the project on time. This proved successful. The design assumption also changed towards the end of print two, as after completing all major functionality works and fixes, the development team was able to make changes to the UI and design features, as well as undertake testing procedures before the final deadline.

Outlined Assumption List

- The project would be relatively easy to build and deploy using a modular framework.

- Team members have a collective unfamiliarity (or very little experience) with the proposed technologies used.

- Each team member had design assumptions/preferences for different design elements

- Assumption that the user base may not be trustworthy, and that there should be a reminder/strike system if books are not returned or running late.

- The initial assumption that team time management would be successful and efficient during the course of the project.

- Assumption that deadlines would not be met towards the end of sprint one.

- Assumption that closer working practices would have to be implemented to achieve milestones/goals - proved true.

- Developed assumption that non essential functionality was considered low priority towards the end of sprint one.

- Assumption that final milestones would not be completed and deadlines would not be met.

- The new assumption that extra/non-essential functionality and improved design would be met towards the end of the final sprint (sprint 2).

2.6   Existing work

Existing work - Stock Management System *A team member has past experience with the design and implementation of a Stock management system. This previous project held many similar design principles and data handling functionality to the SWTlib project.*

Using a modular model, this previous system incorporated 3 nodes (A warehouse, a central office and a shopfront), transferring data between each entity. This proved very similar to the MVC framework that SWTlib is built on.This project also incorporated the use of back end data handling and API's with the entire system being represented by a simple, easy to use User Interface. This architecture and the design features proved to be transferable to the SWTlib project, and aided in the development process.

Existing work - Digital Logbook

Another team member had previously worked on a project for logging maintenance crew issues and fixes. This system uses real time updating and referencing. When treating inventory items in the same domain as these issues, fundamental design features and methods can be transferred from project to project. The design principles form this past project also set a basis for the design principles of SWTlib.

2.7   Reference Work

Reference work consisted of many different tutorials, personal projects and solutions posted freely on open internet message boards. This includes, but is not limited to:

- Youtube personal project walkthroughs and tutorials from a variety channels.

- Solutions posted on Internet forums and message boards.

- Microsoft ASP.NET core online documentation, examples and tutorials.

- Existing work projects.

2.8   Technologies

- Visual Studio Community Edition 2019 - IDE used for developing web apps, programs developing in .NET and ASP.NET core. Used by all team members to develop the SWTlib software.

- Docker - Virtualization SaaS and PaaS used to package and run software. Docker was used by our team to make SWTlib transferable.

- MySQL Workbench - A visual database design tool. Used to visualize, design nd maintain our database.

- C# - General purpose, multi paradigm programming language. Used in the MVC model with ASP.NET Core to enable controllers and construct cshtml pages.

- HTML (cshtml), Javascript and CSS (and external libraries ö bootstrap, chart.js etc.) - General purpose web technologies and libraries. Used to design, construct and style features of the UI.

- JSON - Open-standard file format.

- PHP - General purpose programming language, used for elements of authentification.

- OAuth2 -Access delegation framework, used to grant users access to SWTlib via Gitlab login and university credentials.

- Gitlab - Version control platform and repository for devOps/agile development. Used by all team members to work on SWTlib software in conjunction.

2.9   Web Frameworks

- ASP.NET Core MVC - Web Framework - used by team to build, modulate and maintain SWTlib.

- Entity Framework Core - Object/relational mapping framework. EF Core API used to create database and tables using migration.

2.10   Project Risks

2.11   Project Risk Management

2.12   User Stories

Manage Books

As a SWT chair member, I want to be able to add/remove books to a digital library, so I can curate the content in the library.

Library data can be imported to be added to the database

Easy to use UI, where user can enter book data exists *The story is done when: A user can use the website to add/remove a book entity from the library.*

Library data can be imported to be added to the database

Easy to use UI, where user can enter book data exists

View Books

Rental System

Log in System

Wishlist

Search Function

Reminders System

Statistical Analysis

2.13   Project Organization
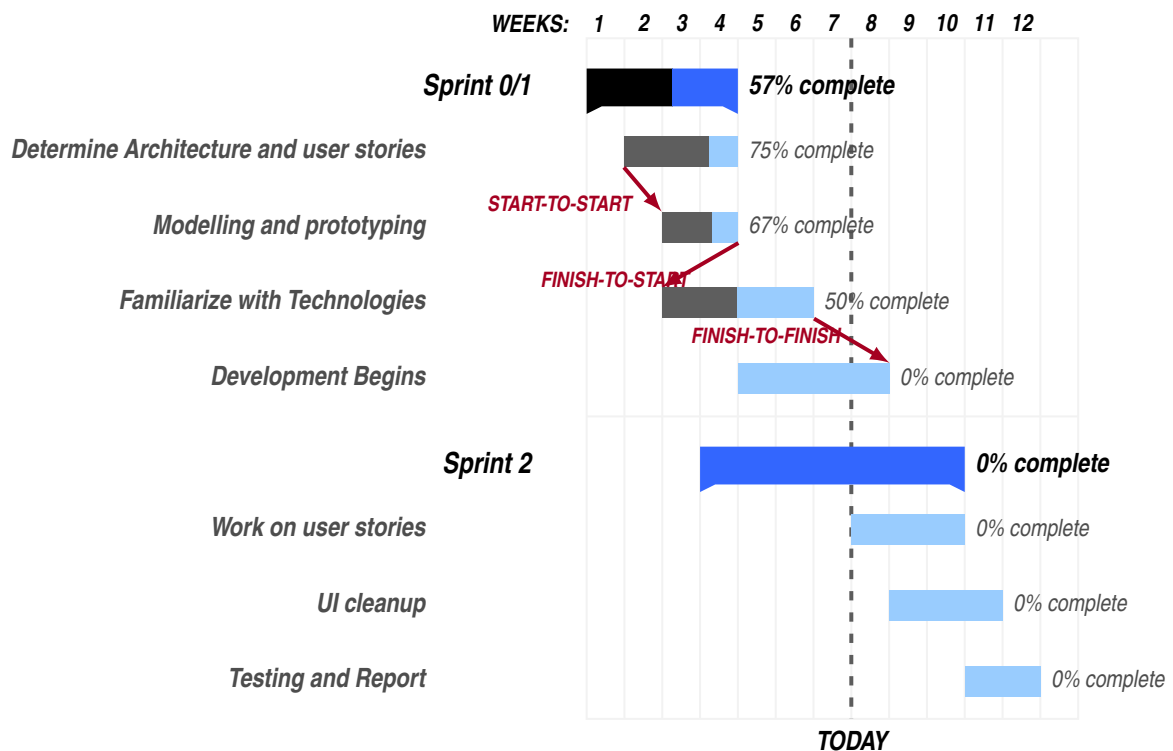
Approximate weighing on mark: 5%
Approximate percentage of expected report length: 5%

Document your project plan and its resulting work packages and milestones. Include a Gantt chart to illustrate the timing and duration of each work package. You may label and reference the work packages and milestones, and refer to them throughout your report. For team projects, state how the work was distributed among the team members, along with each member's responsibilities, contributions, and total worked hours.

In addition, describe your chosen development process, and the tools and technologies that you have applied to your solution.

3   Design

Approximate weighing on mark: 20%
Approximate percentage of expected report length: 25%

Detail and justify the architecture and design of your solutions, such as algorithms and software; the focus depends on your project. Moreover, describe the components within your architecture and their purpose or responsibility. Be clear about the limitations of your design.

Use appropriate diagrams to help communicate your concepts clearly and concisely. Present algorithms as pseudocode, and reference and explain diagrams and pseudocode in the main text of your report.

3.1   Architecture & Design

3.1.1   ASP.NET Core MVC

For our architecture pattern we decided to use Microsoft ASP.NET Core MVC pattern, which is a rich framework to build web-applications. We are using the Core-Version 2.2, which is a stripped-down framework, based on the well-known ASP.NET Full Framework, since it is lightweight and modular. There are no unnecessary and unused packages integrated. Another aspect are the cross-platform needs stated in the requirements. ASP.NET Core works on Microsoft machines as well as on Linux or MacOS. If some additional functionalities which aren't already included into the framework are needed, you can easily import them as NuGet-Packages such as SQL-Connection Services. Since we wanted our application to not only be lightweight but also customizable, we decided to use the MVC pattern. With this approach our web-application could be easily modified and extended, without necessarily changing any logic or components. Another benefit is the possibility to work together on the code without disturbing other developers. Since MVC supports rapid and parallel programming one developer can work on the backend logic while someone else designs the views. The idea was the benefit of faster development, without someone being stuck on essential functionalities of the application. As stated in "Foundation of Software Development" good architecture and design supports later changes and reusability. Also, high cohesion and loose coupling are desirable design principles, which are perfectly fulfilled with the MVC architecture pattern. As our programming language we are using C over F since none of us worked with F before and C is more common. Even though Microsoft recommends in its documentation to use Razor Pages which "makes coding page-focused scenarios easier and more productive" we decided to stick to the basic MVC-model for the sake of clarity regarding components. Unfortunately, we were not able to implement the authentication system without a razor page due to the lack of experience with the used architecture. We were heavily relied on tutorials and documentations which we could only find for Razor Pages. All our attempts to convert the Razor Page to our basic MVC-model failed for some reason. We could not figure out in time how to solve this problem, so we stuck to the working solution.

3.1.2   Entity Framework Core

In addition, we decided on Entity Framework Core for our communication between our application and the database. Entity Framework core is essentially an object-relational mapper (O/RM) which eliminates the need for us to write most of the data-access code. There are two different approaches using Entity Framework Core. The first one is called Database-First, which means that you first create all tables in your database and based on these tables you must create your classes. The second approach is called Code-First, where you create all your domain classes in your code and Entity Framework will create these classes as tables in the database. Since we had no database, we decided to use the Code-First approach. The workflow for the Code-First approach looks like this:

```
1 Create/Modify Domain Classes => Configure Domain Classes
    => Automated/Code-based Migration => Database
```

To create the PrimaryKeys, ForeignKeys, relationships, datatypes, etc. between the several domain classes there are so called predefined conventions in Entity Framework Core. Because of that we usually do not need to do additional configurations but rather tell the Entity Framework API which convention to use. We had trouble to declare many-to-many relationships at first but after some research we found out that Entity Framework Core in contrast to Entity Framework 6 has no convention for many-to-many relationships. The solution was to use Join-Tables and the built in Fluent API to include the foreign keys of two tables to make them composite primary keys. For more details see section "Database". After we created our domain classes and specified them in our LibraryContext.cs class we needed to create a migration for Entity Framework Core. This migration tool automatically updates the database schema if some changes were made to the models. Therefore, you can edit your models without deleting the entire database and losing all you stored data, but rather update it with Entity Framework Core. You can revoke a migration and return to the last successful built.

3.1.3   Database

After talking to AraCom and the Rechenzentrum we decided to use MySQL as our Database. We got provided with a MariaDB which is created by the MySQL company and therefore very similar to MySQL. Our NuGet-Packages and code reference a MySQL-Database which works just fine with MariaDB. We started by implementing all necessary domain classes e.g. book, rental, author, category, keyword. Since we were unexperienced with database models or SQL-databases at all we had to figure out how to create the right relationships between our classes. Due to a hint from Prof. Lüttgen we had a look at database normalization and tried to satisfy the normalforms. For our User-table as well as Author-table you could argue that the names do not satisfy the 1NF, but since we are using the user names only for displaying purposes and rather the user Id for identification, there was no need for us to separate first name and last name. We made the same decision for authors, because there are often authors with a shortened first name like "Brooks, A." what means the first name would not be meaningful. After implementing more and more features or rather requirements we had to redesign our tables and
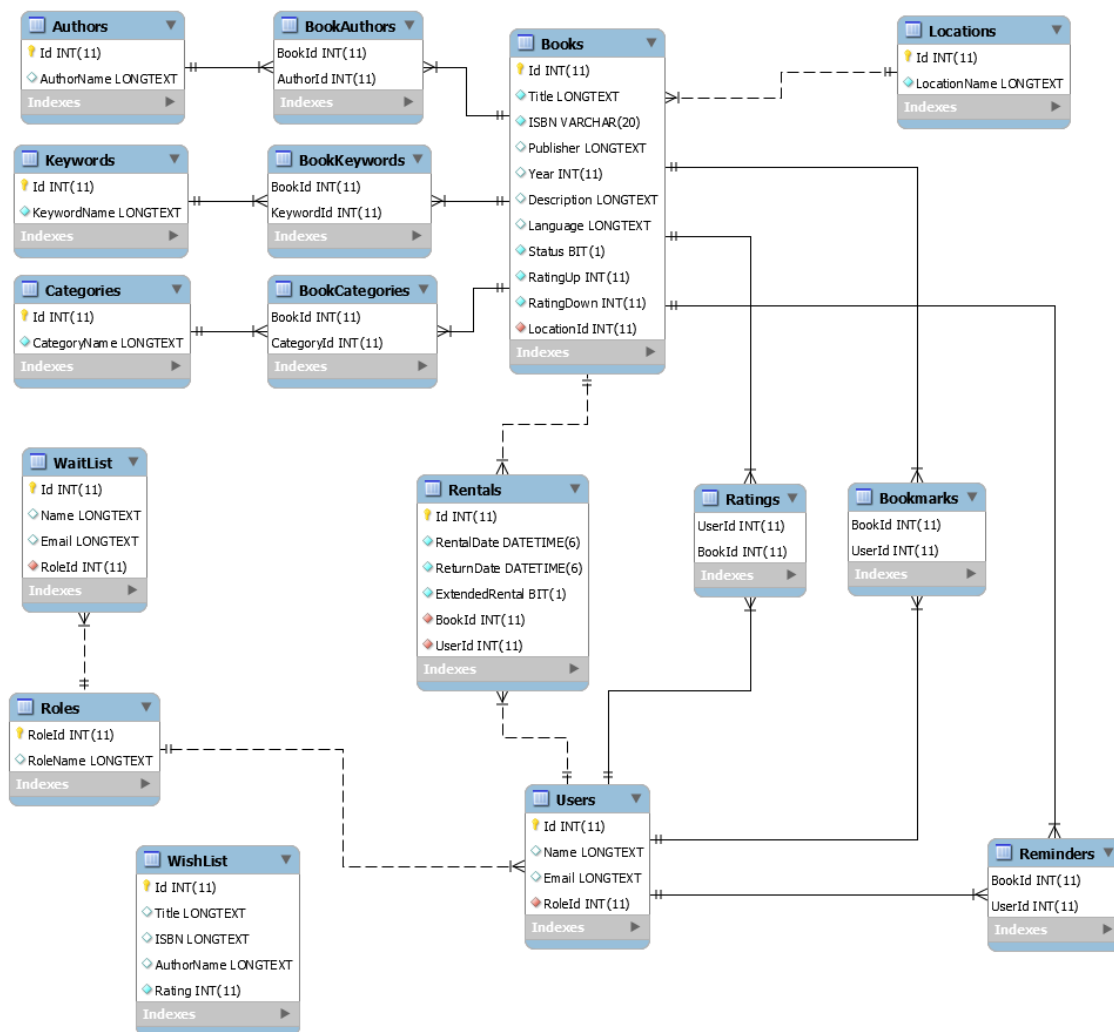
Figure 1: EER Diagram Database

relationships. The final database structure can be seen in following EER-Diagram:

3.1.4   Authentication

For our authentication-system we had the choice between either OAuth2 or Shibboleth. We decided to use OAuth2. You can read more about this in section "4.5 – Log In Implementation Decisions and Obstacles".

3.1.5   Scripts & Styles

After talking to AraCom which architecture we should use we were thinking about using React or Angular for our frontend styling. But since none of us worked with neither the React library or the Angular Framework, AraCom advised us to stick to ASP.NET Core

since it can handle all the frontend sufficiently and it would take us to much time to get familiar with either of these frameworks/libraries. As our base styling we are using the already in ASP.NET Core implemented Bootstrap 4 Framework. To achieve some custom colors for e.g. buttons, navbar, sidebar etc. we are using a custom css file which can be found in "SWTlib/wwwroot/css/site.css". In addition, we use Fontawesome for some icons in our application such as the lens before the searchbar, the bookmark icon as well as the thumbs (up and down) for our rating system. Even though we are mostly unfamiliar with JavaScript and we tried to avoid it as much as possible there were some cases where implementing a feature without JavaScript would have been inconvenient. That is why we used a plugin called "Chosen" for our Add Book form. With this plugin we were able to have multiple selectlists with search function in a more user-friendly styling. For more user feedback we added a busy indicator when loading data, so the user knows the application is doing something. The code is stored in "SWTlib/wwwroot/js/site.js" and within a component which will be explained later.

3.1.6   Software

As our IDE we decided to use the newest Visual Studio 2019 from Microsoft, since it provides every necessary tools to build a web-application based on ASP.NET Core and has an included package manager. We also used Docker for Windows but because some of us do not have Windows Professional or strong enough Computers/Laptops they had to run the code via the "IIS Express" mode comming with Visual Studio. Nevertheless, the application supports Windows Docker-containers and they also can be changed to Linux-containers. As our DevOps we got provided with a GitLab repository from the Rechenzentrum of the University Bamberg. We first ran into some issues regarding the CI/CD pipeline since we never used one nor developed in a team using GitLab or Docker. We managed to get the pipeline working with our branches, but when we create a merge request for the master img:build passes but the deploy:production fails. Due to lack of time we could not fix that issue.

3.2   Components

The solution consists out of two separate projects called "LibraryData" and "SWTlib". The reason is to separate the database context and its entities from the logical layer and views in our application. Therefore, we can change and modify our database and entities without affecting our logical layer too much. For this purpose we are using Entity-Framework Core as stated above.

3.2.1   LibraryData

*The LibraryData project is part of the used entity framework and code first approach. It consists out of all entity-class models used in our application as well as the dbcontext. For the relations between these models please check EER-diagram stated in the section "Database". Each entity is stored in an individual *.cs-class with a representative name in the model folder. To describe these classes I will divide them in by two groups. In*

the first group there are the main entities "Book, Author, Category, Keyword, Location, Rental, Role, User / WaitlistEntry, WishlistEntry". All these entities have a primary key called "Id" and some class specific properties e.g. name, title or duration and some relationship specific properties as you can read in section "Entity Framework Core". In the second group we have "BookAuthor, BookCategory, BookKeyword, Bookmark, Rating, Reminder", which are all somehow join-tables since they have foreign keys as primary keys to bind two entities together. For example, there can never be a Bookmark for a book with Id = 3 without the userId. If u delete a user, all connected tables will be updated and respectively all corresponding entries deleted.

All models have for their properties get and set methods so entity framework can read and manipulate the data in the database. Let us take the "Book"-entity as an example. As you can see in picture 3.1 we declare each property as public, so we can access those properties from outside the class. The integer "Id" tells entity framework to set this property as the classes' primary key. For the properties "Title" and "ISBN" we added an attribute "Required", which tells the entity framework to set them to "NOT NULL" in the database table. As an addition we restricted "ISBN" as a type "varchar(20)" and added a custom client-side input validation, which checks, if a given "ISBN"-String is already existent in the books-table. Another specific property is "Year". Since integers cannot be NULL in the database we needed to declare it as a nullable integer "int?". This will ensure, that books without a publishing year will be added to the database. At line 28-29 we declare a one-to-many relationship between the book-entity and the location-entity which is an integrated convention of entity framework as you can read in section "Entity Framework Core". In the location-entity we have the "ICollection<Book>" which declares the other end of the relationship. Same thing in line 30 for "Rental"-entity, we declared a one-to-one relationship between a book and a rental, so a specific book can be existent in the rental table only once. From line 32 to 35 we declare all join-table relations. Since entity framework core does not support many-to-many relationships by conventions yet, we had to declare them ourselves using the built in fluent-api what we will describe in a moment.

In the "LibraryContext.cs" file we have a class declaration "LibraryContext" which derives from to entity framework class "DbContext". This essentially represents a session with the database which can be used to query and save instances of entities to the database. In that class we declare "DbSet<TEntity> Name" where each TEntity represents an entity in our Models folder. Each individual DbSet represents a new table in the database with the given name created automatically by entity-framework like described in the "Entity Framework Core" section. In line 27 we have a constructor for our LibraryContext options. This is needed so we can use dependency injection for our database connection. The connection to our MySQL server is declared in the Startup.cs file which will be described in another section. From line 29 we are using the fluent-api to override some conventions. The OnModelCreating method from Entity Framework Core allows us to declare our tables "BookAuthor, BookCategory, BookKeyword" as join-tables and therefore as many-to-many relationships between book and the respective entity. With the "HasKey()"-Method we set the "BookId" as well as the respective "EntityId" as a primary key pair. In the respective Entity-files we have declared the "many"-part of the relationship with "ICollection<TEntity> Name ". The other declarations in this file should be self-explanatory since the naming conventions are very clear.

Figure 2: Book Domain Class

In the Migrations folder are all latest migrations stored. In order to keep the database schema in sync with the Entity-Framework Core model you need to create migrations out of the current domain classes. Whenever you change the entities or the DbContext class the Entity-Framework Core API builder needs to create a new migration and apply it to the database. If you have created some errors which are not detected by IntelliSense and your build won't work, you can always remove the last migrations to get back to an old state of your database. It is possible to delete some files of the oldest files if you are confident that the build works, to keep the folder clean. We sometimes had issues that we changed some primary key or foreign key properties and could not apply them to the database. The only way we could find, was to delete the entire tables and recreate them with a new migration. So, it is recommended to create a backup of the existent database before applying any new migrations.

```
27                public LibraryContext(DbContextOptions<LibraryContext> options) : base(options) { }
28
       0 references | 0 exceptions
29    ▣         protected override void OnModelCreating(ModelBuilder modelBuilder)
30              {
31                  modelBuilder.Entity<BookAuthor>()
32                      .HasKey(c => new { c.BookId, c.AuthorId });
33
34                  modelBuilder.Entity<BookCategory>()
35                      .HasKey(c => new { c.BookId, c.CategoryId });
36
37                  modelBuilder.Entity<BookKeyword>()
38                      .HasKey(c => new { c.BookId, c.KeywordId });
39
40                  modelBuilder.Entity<Book>()
41                      .HasIndex(s => s.ISBN)
42                      .IsUnique();
43
44                  modelBuilder.Entity<Book>()
45                      .Property(s => s.Status)
46                      .HasDefaultValue(false);
47
48                  modelBuilder.Entity<Book>()
49                      .Property(s => s.LocationId)
50                      .HasDefaultValue(0);
51
52                  modelBuilder.Entity<Rental>()
53                      .Property(s => s.ExtendedRental)
54                      .HasDefaultValue(false);
55
56                  modelBuilder.Entity<Bookmark>()
57                      .HasKey(c => new { c.UserId, c.BookId });
58
```

Figure 3: ModelBuilder in LibraryContext

3.2.2 SWTlib

The SWTlib project is structured by the Model-View-Controller pattern and stores all the logic and ui components. Since we are using dependency injection for our DbContext we need to create an instance of the "LibraryContext" in each controller. Let us begin with the backend and logic part:

Startup.cs: *From Microsoft Docs "The Startup class configures services and the app's request pipeline." In here we declare all needed services such as our SQL-connection and our authentication service. So, for authentication system we need to configure a few services inside our Startup.cs file. We create a session by using cookies and we define them as essential. After that we define our AuthenticationScheme which is needed to identify a user as you can read in the controller section. We add the cookie and create an OAuth authentication and populate it with data we retrieve from the GitLab login such as the ClientId and the ClientSecret as well as the Id, Name, Avatar and Email assigned to that GitLab account. We can later access those data with the HttpContextAccessor we define in line 96.*

For our database connection we add a service called "AddDbContextPool" and receive our connection string from the "appsettings.json" where all information's for the database login e.g. server name, password, port, etc. are stored.

```
95                    services.AddSingleton(Configuration);
96                    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
97                    services.AddDbContextPool<LibraryContext>(
98                        options => options.UseMySql(Configuration.GetConnectionString("LibraryConnection")));
99                }
```
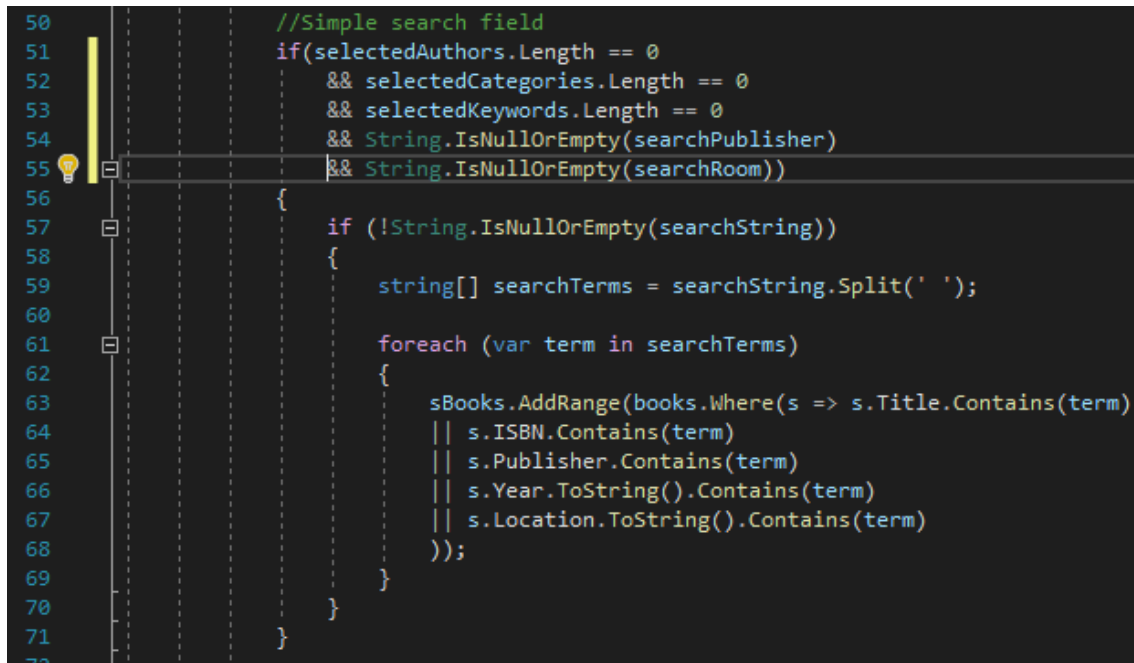
Figure 4: Startup.cs

Controllers:
We are not using a separate controller for each entity in our LibraryContext models but rather tried to implement the ActionMethods logically into controllers such as the "Book-Controller" does not only store the CRUD operations for the book entity but also the logic for the rating system as well as the xml-import logic. But we will get to this later. There is a total of twelve Controllers where describing every single one would not make much sense. That is why we picked the most important ones to cover the essential logic used. We begin with the "AccountController" since this is where our application logically starts.

AccountController: *As stated below we first must create an instance of our LibraryContext and pass it to our controller constructor. In line 15-20 we have the Login ActionMethod which is a HttpGet request which returns a ChallengeResult and challenges the given authentication schemes' handler. This is necessary to identify the user. If the user is identified, he will be redirected to the "returnUrl" which is our Dashboard page. The Logout method clears the session and redirects the user to the logout page of GitLab. By clearing the session, the user is no longer logged in and must again verify his identity.*

AdminController: *In this controller we store the logic for the admin role which integrates the role management feature into our application. In line 26-32 we have the Index ActionMethod which first stores each entry from the WaitList-table in the database into list. Then we call an own method we called "RoleDropDownList" to populate a selectlist with role items. In the end we pass the list to the view. The second Action is called Approve and takes two inputs "userid" and "roleid". The view passes these to integer values to the controller respectively the Approve method. The method is a HttpPost method which will update the database if the modelstate is valid. First it queries the database for an entry in the WaitList-table with the Id == userid and stores the result into a variable "entry". After checking "ModelState.IsValid" it will create a new User-entity called "user" and declares its properties e.g. the "Name"-value of the new User-entity equals to the queried Name-value of "entry" (entry.Name). Same for "Id" and "Email". For "roleId" it should take the new integer passed from the view. After that the new created user is being added to the users table and the old user "entry" is being removed from the WaitList table.*

SearchController: *This controller stores the logic for a normal search as well as the logic for the advanced search functionality. The Results ActionMethod takes several input variables depending if you use the "normal" or the "advanced" search. For the normal search, the entire string from the form is passed to the "searchString" variable. In line 51 we check if each other string or array of strings is empty, so we know it is passed by the "normal" search. After verifying our searchString is not empty we split that string separated by spaces and store each term into an array called "searchTerms" (line 59). After that we check the database for each term and store the corresponding book in a list of book-entities. That list is then passed to the view. This means we search the database*

for each word individually and not by the entire string. We could have added another "and"-clause in our loop to search for combinations of words, but we decided to keep it that way since the database would not be filled with to many books and this would also prevent some possible typos in the search form.

```
50              //Simple search field
51              if(selectedAuthors.Length == 0
52                  && selectedCategories.Length == 0
53                  && selectedKeywords.Length == 0
54                  && String.IsNullOrEmpty(searchPublisher)
55                  && String.IsNullOrEmpty(searchRoom))
56              {
57                  if (!String.IsNullOrEmpty(searchString))
58                  {
59                      string[] searchTerms = searchString.Split(' ');
60
61                      foreach (var term in searchTerms)
62                      {
63                          sBooks.AddRange(books.Where(s => s.Title.Contains(term)
64                          || s.ISBN.Contains(term)
65                          || s.Publisher.Contains(term)
66                          || s.Year.ToString().Contains(term)
67                          || s.Location.ToString().Contains(term)
68                          ));
69                      }
70                  }
71              }
```

Figure 5: Normal Search

The advanced search works similar to the normal search. For the Authors, Categories and Keywords we are using a HashSet of Books, so if the algorithm finds the same book twice depending on its search value, it will be added only once to the list.

HomeController: *Within this controller we store the logic for our landing page e.g. the Dashboard as well as for the logic for the shared "AddContent" page and the create and delete operations for our room e.g. location entity. It also requests all Categories from the database together with the corresponding books. This data is passed to the view to display the Category-tiles on the Dashboard.*

BookController: *This controller is a little bit more complex, since it has the most relations to other domain classes e.g. authors, categories, etc. The ActionMethod "Index" is used to display a list of all books available in the database. For this purpose, we created a new ViewModel which will be explained later in the section "Models". This "BookViewModel" stores each corresponding value from the database into a list of "BookViewModel"-entities. This list is then passed to the index-view. The "Details"-Method takes as an input value a book-Id and searches the books-table in the database for that Id. After that we call a custom method called "CheckRating" (line 93-106) with the given bookId and a userId, which we get from the current session with these two lines:*

The method sets a "ViewBag.Ratings" to either true or false, depending if it finds an en-

```
69              var sessionId = HttpContext.Session.GetInt32("_Id");
70              var user = _context.Users.FirstOrDefault(i => i.Id == sessionId);
```

Figure 6: Get userId from session

try matching the two input parameters. This logic will later been taken from the view to enable or disable the rating function for the given user.

```
93              public void CheckRating(int? BookId, int? UserId)
94              {
95
96                  var rating = _context.Ratings.Find(UserId, BookId);
97
98                  if (rating != null)
99                  {
100                      ViewBag.Ratings = true;
101                  }
102                  else
103                  {
104                      ViewBag.Ratings = false;
105                  }
106             }
```

Figure 7: Rating Logic

The "Create"-Method consists of a get-request and a post-request. The get method retrieves all the necessary data from the database e.g. to populate the dropdownlist for locations by using the custom method "LocationDropDownList" as well as get all available authors, categories and keywords. Let us have a look at how we get a MultiSelectList for the Author-entity. First of all, we declare a variable named authors. From our authors in our context we select each author Id and its corresponding authorName and store it as a new item into the variable authors. We now have a list of items which we pass to the ViewBag.Authors as a new MultiSelectList. We do the same thing for Categories and Keywords (respective other variable and ViewBag names).

```
292             LocationDropDownList();
293
294             var authors = _context.Authors.Select(c => new
295             {
296                 AuthorId = c.Id,
297                 c.AuthorName
298             }).ToList();
299             ViewBag.Authors = new MultiSelectList(authors, "AuthorId", "AuthorName");
```

Figure 8: Create MultiSelectList

After submitting the form from the view, the HttpPost Create-method will be called. It takes the given Book model as well as the corresponding arrays of strings for author,

category and keyword as an input. The arrays can be filled with Ids, if the author already existed or with names if the author is not existent in the database. To verify this, we need to try to parse each item in the array to an integer. If the parse returns true, we can convert the Id-string into an integer and store it in a new List of integers called "authorsList". If the parse returns false, we know the author is non-existent in our database and we have to create a new entry. We do this in line 339 where we set the AuthorName to the value of the item. After adding the new Author to the database and saving, we need to request Id from that new Author and add it to the "authorsList". The last thing to do is to add the relations to the BookAuthor join-table. To do so, we create a new List of BookAuthor-entities and create a new entry for each item in our "authorsList" where the bookId is always the same Id of the book that will be created and the authorId equals the current item from the "authorsList". We do the same thing for Keywords. Since we do not allow to create new categories within the "Add Book" view, you can only select available categories. Therefore, we only need the join-table logic for categories.

```
324          if (selectedAuthors != null)
325          {
326              //Create a new List<int> to store the AuthorId's.
327              var authorsList = new List<int>();
328
329              foreach (var item in selectedAuthors)
330              {
331                  bool isInt = int.TryParse(item, out int n);      //Try to parse the string item from selectedAuthors to int and return a bool.
332                  if (isInt == true)
333                  {
334                      authorsList.Add(int.Parse(item));            //If success: add item to authorsList.
335                  }
336
337                  if (isInt == false)                              //If false: create a new author and add it to the database.
338                  {
339                      var newAuthor = new Author { AuthorName = item };
340                      _context.Authors.Add(newAuthor);
341                      await _context.SaveChangesAsync();
342
343                      var newAuthorId = _context.Authors.Find(newAuthor.Id);      //Get the Id of the created author and add it to authorsList.
344                      authorsList.Add(newAuthorId.Id);
345                  }
346              }
347
348              book.BookAuthors = new List<BookAuthor>();
349              //Add Book and Author to Join-Table
350              foreach (var author in authorsList)
351              {
352                  var authorToAdd = new BookAuthor { BookId = book.Id, AuthorId = author };
353                  book.BookAuthors.Add(authorToAdd);
354              }
355          }
```

Figure 9: Create Join-Table entries

Since the "Edit" get-method works mainly the same as the "Create" get-method we are going to skip this and jump directly to the post-method. We are now awaiting an Id and three arrays as input. First, we search the books table for the book entry with the given id and store it in a new variable. After that we try to update that book with the new values passed by the form and if that returns true, we call some custom methods to update the join-tables as well. These custom methods take two inputs, an array of the selected values from the form and the book entity. In those methods we create a new HashSet as before and add the past values. Then we need to create a list with the current values stored in the join-table. Now we check if the value pair already exists and if not, we create it as well as delete any entries we do not want in the table anymore, since we updated the book. The "ListEdit" and "ListDelete" methods are only to create a view with a dropdownlist of books. Since the "Delete" method is pretty generic we are going to skip that, but since we are using Entity Framework Core we just need to delete a book we want and EF Core will delete each corresponding entry in the database like all connected

entries in the Join-Tables etc. Next we have the logic for our rating system. We created two ActionMethods "AddUp" and "AddDown". By taking a bookId and a userId as an input, we can search the database for the respective entry and add a new rating to the rating-table in the database. We do this to ensure that each User can rate a book only once. After the rating is added, we count +1 for either "Up" or "Down at the respective book entry. The import XML part in the controller was just a small addition and testing. This could be improved if the feature is wanted. But since it was not stated in the requirements and the xml provided should only be imported once, we did not implement the code we used.

__RentalController:__   In this controller we store all the rental logic as well as the reminders logic. At first, we had a datepicker in the view, but we wanted the duration to be set with a dropdownlist from one to four weeks. That is why we first needed to create model which is stored under " /Models/Helpers/ReturnDateHelper.cs". We use that model in a custom method called "ReturnDateDropDown" to create a list of ReturnDateHelper-entities and store them in a list. The entity has a datetime value and a string property. After that we create a ViewBag.ReturnDate as a new SelectList filled with our list. This represents now our DropDownList as replaces the datepicker. We do the same thing for the method "ExtendReturnDateDropDown". But in addition, we take a RentalId as an input, to search the database for the rental we want do update. This method is called in the "Edit" post-method. Since we want every user to be able to extend the rental just once, we also change a Boolean inside the rental to true. If that Boolean is true, the ExtendRental button inside the view will be disabled. The create method works similar to the book create method, whereas in rental we need to add a userId which we request from the current session. Also, we set the current status of a book to true, which means the book is not available. If a book is being returned, we call the delete method which will simply delete the entry from the database and as seen before, we change the status of the given book back to false. If a book is already borrowed, we wanted to be able to set a reminder for that book. The "AddReminder" and "DeleteReminder" methods take a bookId and a userId and create/delete entries in the given table respectively the inputs. The last two ActionMethods are part of the ViewComponents which will be described in the "Models" section.

__WishListController:__   The WishList page consists of two main parts: A ViewComponent which displays all current wishes from the database and a form to create a new wish and a button to generate an email. The method "Index" is responsible to populate the body of the email. First, we store all wishes in a variable and define a counter. Here is the place to change the output message for the email. We create a string starting with hardcoded text, followed by a foreach loop to concatenate the first five wishes to our string. After that we pass it with a ViewBag to our view.

Then we again have the logic for a ViewComponent which we describe in the next section. The create method checks first the highest rating available in the wishlist table. Higher rating means lower in the list. When creating a new wish, it adds one to the highest value which means the new entry will be at the bottom of the list. This means the first entry always has a rating = 0 and the last entry has a rating = (number of entries). If you want to delete one entry it takes all entries below that given entry and removes one from the entry-ratings.

```
19        public IActionResult Index()
20        {
21            var wishes = _context.WishList.ToList();
22            int i = 1;
23            string message = "Hallo Frau XXZ,%0D%0A%0D%0A bitte bestellen Sie folgende Bücher:%0D%0A";
24
25            foreach (var wish in wishes)
26            {
27                if(i <= 5)
28                {
29                    message += i + ") " + "Title: " + wish.Title + "    ISBN: " + wish.ISBN + "    Author(s): " + wish.AuthorName + "%0D%0A";
30                    i++;
31                }
32            }
33
34            message += "%0D%0A%0D%0AVielen Dank.";
35
36            ViewBag.Wishlist = message;
37
38            return View();
39        }
```

Figure 10: WishListController

The "ArrowUp / -Down" methods basically just change the value of and its adjacent one depending if you push the up or down arrow. They simply switch their rating-values.

Models:
Since our models are separated and stored inside the "LibraryData" project, the models folder in "SWTlib" project contains only Helpers, ViewComponents and ViewModels. We already covered helpers in section controller/rentalcontroller so let us continue with the ViewModels.

ViewModels: *ViewModels are used to combine data from different sources into one model or to cut some data out. It defers from the domain model in the manner that it is a model for the view. So, you can declare each property inside this ViewModel you want to use or to see on your view page. We are using ViewModels for our books, rentals and bookmarks. Since they are very similar to domain models it should be clear how to create and use them.*

ViewComponents: *To be able to display data in a shared view we had two options. Either use PartialViews or the more powerful ViewComponents. We decided on View-Components, since it is the recommended method for our purpose. They look similar to controllers but for us they have only one method called "InvokeAsync". MyRentalsExpire-ViewComponent: This ViewComponent shall display all rentals of the current user which expire in the next seven days or less. To do this we first query all rentals of a given user into a list and then filter them by the wanted date range. The result will be passed to the corresponding view. The other ViewComponents all have a similar logic, requesting any data from the database and passing it to their view.*

Views:
The views represent the frontend of the application. Inside the View folder are other folders which match the controllers. So, for each controller there should be a view folder. In addition, there is a shared folder which stores all shared views such as the layout view, partial views and component views. Inside these view folders are some cshtml files such as "Index" or "Create". These files will be called by the corresponding controller. Since

most of these files look the same, we will show just a few with some special logic.

/Views/Book/Index: In the first line we declare "BookViewModel" as our model which means the view expects to get a "BookViewModel"-entity from the controller. After the breadcrumb navigation we create a foreach loop to query through all books in our model. This will create a div inside the view for each book in the given booklist passed by the controller. To navigate to the details, view of a book we made the entire div a clickable link which will call the "asp-action='Details'" inside the BookController and pass the Id of the current item e.g. the clicked item. For the Bookmarks to be displayed correctly we needed to make an if-else condition. If there already exists an entry in the database with the given bookId and userId, the bookmark icon should be filled. Otherwise it should be blank. Depending on the state of the bookmark a click on the icon should set or delete the bookmark. By referring to a specific asp-action and redirecting back to the page, setting or deleting a bookmark should feel convenient and fast. The quick rental button at the side of the div is managed by an if-else condition which disables the button, if the book status is set to true, which means already borrowed. By pressing the button if active you will be redirected to the "Create" method inside the "Rentals" controller.

/Views/Book/Create: Creating a book brought some issues we needed to solve in order to achieve our goals. We thought in our prototype of searchable multi select lists for authors, categories and keywords. To achieve this, we needed to add a plugin called Chosen-plugin to our code. Chosen plugin is based on jQuery and modifies a given MultiSelectList from ASP.NET Core. After watching a lot of tutorials, we managed to get it working as intended, but we did not think of how to deal with creating new values in the given table e.g. author. The most convenient way would be, if the author would be created automatically while creating the book. To achieve this, we needed to change our backend code as you can read under section "controllers" and we needed to change our jQuery code to allow "create_option" e.g. line 139.

/Views/Shared/...:   In the shared folder are all views which should be available for the entire application such as the layout or components. Inside the Components folder we have other folders named after their respective ViewComponents and inside these folders one Defaul.cshtml each. ASP.NET Core will search those folders for the respective view and that's why the naming is crucial. Inside this cshtml we can use classic html and access the data passed by the ViewComponent. The _Layout.cshtml is our main html file. We declare a head, the needed stylesheets and scripts as well as the different navigations and sidebars. In the top navigation we render the Title of each subpage and on the right side we request the Name and the Avatar from the current session. For the left side navigation, we use simple a-tags to route to the action inside a specific controller. At the bottom of the list we call a ViewComponent. With that component it is possible to dynamically, based on the user's role, show or hide specific navigation items. In that case we only want a user to be able to add content, if he is at least a chair member and to manage role the user must be an administrator. With "@RenderBody()" we tell ASP.NET Core where to render all other views. So, all view outside this shared folder will be rendered at this position. For the right navigation we are using jQuery to display the current time and date and below the two components "MyRentalsExpire" and "Reminder". To give the user feedback about the current state of the application we also added a partial view called "BusyIndicatorPartial" which essentially shows a loading circle over the "RenderBody" part.

4   Implementation

Even though the development team made all the important decisions regarding the direction of the project together, often times the concrete implementation was up to the individual team members.When dealing with various obstacles over the course of our project, the team would try to figure them out by consulting each other, AraCom or via pair programming. Parts of the implementation details are already covered in section 3.

4.1   Primary Development Platform Decision and Obstacles

During the introductory meeting, we were informed that employees of the AraCom IT Service AG would be our advisors for this project. Since they were the ones who would help us in all matters related to technology, implementation and software engineering techniques, their area of expertise was an important factor in our technology related decisions. When we learned that most of AraComs' work is related to either .NET and java, the choice for our main platform was between these tow options.

The team discussed that we were more familiar with java overall, however after some consideration, due to several circumstances we decided that .NET would be the superior choice for us. For one, our team was very inexperienced overall,to a point were some team members had little to no programming experience in general, which means that the familiarity gap between java and .NET was not especially large. On top of that, the team wanted to take the opportunity to try something new - having AraCom as a consulting option presented a chance to use their insight for a different approach than what is usually offered at the University of Bamberg. The vast majority of teaching offerings at the University of Bamberg are focused on java after all. Additionally, AraCom suggested that ASP.NET would ultimately be the most straightforward solution for a team with our lack of experience, as it is designed with exactly the type of application we were building in mind. Finally, the member of our team with the most programming experience in currently with .NET in other capabilities, so working with the same technology in parallel would end up being convenient for him.

With that sorted out, the next decision was what type of .NET we were going to use, as the .NET ecosystem offered us multiple options. Our two main options were .NET Framework and .NET Core. Since one of the goals we had for our application was, that we wanted to achieve cross platform compatibility, the superior choice was to use .NET Core as its runtime (CoreCLR) and libraries are cross-platform. Another major advantage of .NET Core is the very high scalability it offers, because we wanted to design a responsive and performant system, that offered a good user experience even when the number of users goes up.

4.2   Language Decisions and Obstacles

Even though the software was going to be employed a German university, the development team decided that the initial language of the website was going to be English. This

was convenient because two of are team members are exchange students and therefore not fluent in German. The team estimated that English was going to be sufficient, as our main user base was going to be SWT chair members and students. It can be expected that any member or student at the SWT chair is able to speak English due to the fact that the teaching language of SWT courses is English anyway. This approach has the advantage of automatically allowing international students access, too. Adding a switch language option was a low priority consideration at the beginning of the project however it was quickly discarded in favor of more pressing matters.

4.3 Platform Decisions and Obstacles

In the first introductory meeting it was hinted, that mobile compatibility would be an important aspect of the app, as tablets such as ipads are playing a bigger and bigger role in the in the work life of chair members. Taking this into account, the team discussed wether developing a mobile friendly web app or a native mobile app would be the superior option for our project. After talking to AraCom about his issue, we realized that the scope of the project will not allow for the development of a separate mobile app in the 8 weeks we had available. The increased effort would not be wort the relatively minor benefits of such an approach. A mobile friendly web app however would have multiple advantages. No tedious approval process for mobile stores would be required. A web app would be easier to maintain and update. This was an important consideration for us since we would have to hand over the software to others at the end of the semester, which meant that easy and convenient maintainability had to be taken into account. Most importantly though, the team has more experience working with web apps compared to making a standalone mobile application.

In light of these arguments, the team thought it was appropriate to use a "mobile first" approach. However we quickly discovered that this was unrealistic as well. As time became a more and more pressing issue and there were already more than enough problems with the desktop version of the website, the mobile aspect of the web application faded more and more into the background. Nonetheless, throughout the project there was some effort to keep UI elements scalable so the site would stay relatively mobile friendly.

As for compatibility with different browsers, some of the students in our group used Mozilla Firefox, others on the other hand used Google Chrome. This would ensure that the web app would be fully tested and functional for at least these two important web browsers.

4.4 Database Decisions and Obstacles

In terms of the database management system, the team decided to go with MariaDB over a regular MySql database software, convinced by better performance, optimization as well as a bigger set of features. Overall, we did not spend a substantial amount of time and research to reach this conclusion. That is because, since we were going to perform only relatively simple operations on our database the exact choice here was not especially important.

4.5   Log in Implementation Decisions and Obstacles

The main goal of the log in system was that users should be able to use their existing University of Bamberg accounts with their given ba-number and password as credentials. In order to accomplish this there were two options we could have chosen. One option was to use Shiboleth as a single sign on service. This is the middleware that is for example used in the log in system of the virtual campus service of the university. Regarding this, we received advice by the Rechenzentrum of the University of Bamberg, that using Shiboleth for our log in would complicate matters a lot and that the additional features that Shiboleth offers compared to the alternative are not necessarily useful for our application.

Said alternative is OAuth2, a protocol that would allow us to use user the Gitlab server of the University of Bamberg as a Authorization server. Our aids from AraCom similarly said they had experience with asp.net applications and OAuth2 and postulated that it would be easy to integrate OAuth2 with our application with ASP.NET. The oAUth2 flow used in our application works as follows:

First, the application requests to authorization endpoint of the Gitlab server of the University of Bamberg to authorize the user. Then the user is asked the question if they wish to authorize the application to access their information. If the user accepts, the user is send to a redirect URI with a temporary access code. The app then uses the token endpoint of the Gitlab server to get a permanent access token which can now be used to access any resources provided by the Gitlab API. Concretely the important parts of the implementation look like this. In the startup.cs file this call to the AddAuthentication method

```
1   services.AddAuthentication(options =>
2               {
3                   options.DefaultAuthenticateScheme =
                        CookieAuthenticationDefaults.
                        AuthenticationScheme;
4                   options.DefaultSignInScheme =
                        CookieAuthenticationDefaults.
                        AuthenticationScheme;
5                   options.DefaultChallengeScheme = "Gitlab"
                        ;
6               })
```

registers the authentication services. Here we also make sure the authentication Information is saved in a cookie.

```
1   .AddOAuth("Gitlab", options =>
2                   {
3                       options.ClientId = "xxx";
4                       options.ClientSecret = "xxx";
5
6                       options.CallbackPath = new PathString
                            ("/signin-gitlab");
```

```
7
8                              options.AuthorizationEndpoint = "
                                 https://gitlab.rz.uni-bamberg.de/
                                 oauth/authorize";
9                              options.TokenEndpoint = "https://
                                 gitlab.rz.uni-bamberg.de/oauth/
                                 token";
10                             options.UserInformationEndpoint = "
                                 https://gitlab.rz.uni-bamberg.de/
                                 api/v4/user";
```

afterwards the OAuth authentication handler is registered by calling the AddOAuth() method. We also give the information needed to register the application, which includes the ClientId, ClientSecret and CallbackPath.

Here we also specify the server endpoints we need to obtain the authorization token as well as the user information endpoint, which provides the user information we need if requested, The user information endpoint gives us the following JSON response from the Gitlab server:

```
1  {
2    "id": 527,
3    "name": "Simon Alexander Nuetzel",
4    "username": "simon-alexander.nuetzel",
5    "state": "active",
6    "avatar_url": "https://secure.gravatar.com/avatar/
        ec28c488e2ff77a2d13f93fa83dafe61?s=80&d=identicon",
7    "web_url": "https://gitlab.rz.uni-bamberg.de/simon-
        alexander.nuetzel",
8    "created_at": "2018-10-17T12:36:23.205Z",
9    ...
10   "last_sign_in_at": "2019-07-27T14:37:18.946Z",
11   "confirmed_at": "2018-10-17T12:36:23.135Z",
12   "last_activity_on": "2019-07-27",
13   "email": "simon-alexander.nuetzel@stud.uni-bamberg.de",
14   ...
15  }
```

We only need a small part of the information we have available. For our application, we grab the user id, the user name, the email and the Gitlab avatar from this response:

```
1  options.ClaimActions.MapJsonKey(ClaimTypes.
     NameIdentifier, "id");
2                              options.ClaimActions.MapJsonKey(
                                 ClaimTypes.Name, "name");
3                              options.ClaimActions.MapJsonKey("urn:
                                 gitlab:avatar", "avatar_url");
4                              options.ClaimActions.MapJsonKey(
                                 ClaimTypes.Email, "email");
```

and store these values in our IndexModel class to make them accessible. Additionaly, we store data we need in a Session so we can access it throughout, e.g for always displaying the user name when logged in or for assigning rented book to a specific ID.

```
1   GitlabName = User.FindFirst(c => c.Type == ClaimTypes.
        Name)?.Value;
2                   GitlabId = User.FindFirst(c => c.Type ==
                        ClaimTypes.NameIdentifier)?.Value;
3                   GitlabAvatar = User.FindFirst(c => c.Type
                        == "urn:gitlab:avatar")?.Value;
4                   GitlabEmail = User.FindFirst(c => c.Type
                        == ClaimTypes.Email)?.Value;
```

One major obstacle we were unable to resolve was that We were unable to simply fit the log in page into our MVC model. Therefore we used a separate Razor Page. Razor Pages are compatible with the standard ASP.NET MVC, however there is some downside to this is that is makes the structure of the project less concise. Another problem we faced due to this constellation was users were temporarily directed to the log in page whenever they tried to re-load the home page.

4.6   Role Decisions and Obstacles

The users of our application were supposed to be categorized in three different roles with different access rights: "Admin", "ChairMember" and "Student". For assigning the roles to the different users, we decided to go with the following implementation:

When a user tries to sign in, the application creates a user object with the user's information and then attempts to save this user in the user table of our database. If this user already exists in the table, the user is supposed to be logged into the system and get access to the site according to their specific role. However, if the user has not been added to the pool of users and assigned a role, the user is instead added to a wait list table and the access to the site is limited. Within the website, users whose role is set to "Admin" can access the wait list, assign a role and approve individual users. These users are then added to the user table so they are recognized as approved users for the following log in attempts.

When implementing this, we encountered problems when assigning users to the correct table both when initially signing into the system and when the admin tries to move users from the wait list to the user table.

5 Evaluation

In Terms of Quality Assurance we started at the fundamental. When creating our User Stories we tried to fit the INVEST and SMART -criteria

INVEST and SMART -criteria

<u>INVEST-criteria</u>

INVEST stands for "independent", "negotiable", "valuable", "estimable", "small" and "testable".

The first Criteria "Independent" states, that a User story should never be dependant from another User Story from the same sprint. This is supposed to give more flexibility for the group as they do not have to worry about those dependencies.However it is not entirely possible to strictly separate certain tasks completely, therefor we made sure that everyone knows about them so that if in fact issues in that regard appear, the team was able to shedule their tasks accordingly.

The second Criteria "negotiable" states, that no explicit information about the design/architecture of the project is stated, but merely functionality only.

The third Criteria "valuable" states, that every User Story is supposed to be worth for the Customer. This shall give them more of an idea and understanding of what we are actually working on and letting him in on the project.

The fourth criteria "estimable" states, that a team is able to estimate the amount of time/resources needed to finish a User Story.This in particular was useful to us, as it allowed us to prioritize our User Stories better and therefor plan our sprints more efficiently.

The fith Criteria "small" states, that a User Stories size should take, at max, a third of the Sprints Time to implement. At first we did not apply this to our user stories, resulting in only a few large User Stories which would have totally bloated the scope of a single Person. Therefor we decided to break those User Stories down into smaller ones, allowing us to distribute the workload a lot more efficently

The sixth and last Criteria "testable" states, that it should be possible to write tests for the story's acceptance criteria. This opens up the possibility of writing tests before implementing the user story, which is a great way of assuring the quality of the software. We did not really take testability into account when creating the user stories, because we considered a user story completed when all the associated tasks were done.

<u>SMART-Criteria</u>

The SMART criteria are a quality assurance technique for tasks and it stands for specific, measurable, achievable, relevant and time-boxed.

The "Specific" Criteria states, that every team member should be able to understand the task and its Context. This helps clearing misunderstandings and avoids overlapping tasks or even implementing tasks wrong. When we came up with our tasks, the whole team was present to make clear, that everyone understands the content and context of the task. Although the language barrier was a problem at first, we got used to it quickly and were able to make everything "Specific".

The Criteria "measurable" states, that the team can decide if task is done and can be moved forward, or is not done, hence need some more work to do. This is especially crucial as a task might be done in the eyes of a team member, however another one does not think so and continues to work on it, wasting time and resources for nothing and slowing down the progress.

The "achievable" Criteria states, that a task is supposed to be doable for the person who is assigned to it. This was especially important for us, as we had a group mixed with the most diverse members in regards to Coding experience. We needed to make sure that everyone was able to do his tasks, or else other members would need to shoulder more work then usually planned.

The "relevant" Criteria states, that a certain task needs to be relevant in any way or form to the User Story it is assigned to. Having tasks that are irrelevant to the User Story would only end up in confusion about their context. Considering that all of our tasks were created in order to work on a User Story, most of our tasks for relevant, with some exceptions of course.

The "Time-boxed criteria" states, that a task should be limited to a specific amount of time in which the task owner is trying to solve it. If he for some reason can not do it in the given time, the task owner should seek for help in order so the that the problem can be solved faster rather than having one developer being stuck. In our case we did not really make use of this criteria, as most of us are quite unexperienced when it comes to Coding so therefor we stayed in Contact from the very beginning.

5.1 Acceptance testing

All of these tests were done on a device that Runs 10 Machine using the Browsers Google Chrome and Firefox as these are the 2 most commonly used browsers.

Our Program was developed with the needs of the Customers and Clients needs in mind. Those needs we took from what was given to us in the Project brief. So in Sprint 0 we Started to analyze the Project brief and extracted functional and non-functional Requirements (See 1.2. Methodology) from it and refined them in sprint 1. When we thought a requirement was fully implemented, we double checked that with the Fit Criterion and examined whether they were met or not. Also we repeated this every time a major update was introduced to our system, in order to see if that might have broken an already existing feature or how the new features blends in with the previously implemented features.

In addition we also thought of Some realistic use case scenarios and Simulated them multiple times with slight variations to see if this is going to crash the system or not. This allowed us to see if and where errors have been made, so that we can tackle them directly. By doing so, we were able to make sure that the key functionality of our product is going to a free-of-bugs experience for the User.

A better solution to this would be a Test-driven development. This method would have allowed us to make sure that every requirement would have been met, by turning them into test-cases before the development starts. By doing so, you can work directly towards the goal of meeting the tests and if all tests are successfully you know that the requirement has been fullfilled. Despite these advantages we decided not to go with the

Test-driven development due to our inexperience in Software engineering and programming and therefore our lack of confident to correctly asses the requirements and then write proper test-cases beforehand.

5.2   GUI Testing

In Order to make sure that our Design was satisfying for the Costumer, we decided to Create an Interactive PowerPoint presentation featuring our idea of the Design. After presenting it, taking in the feedback from the Customer and correct the PowerPoint, so that it would fit the demands of said Customer. Soon after we started with the Implementation of our Graphical User Interface with our prototype in mind. By staying as close to the Prototype as possible we were able to secure the satisfaction of the Client, as he already seen the design and gave feedback for improvement. Additionally, by planning the design beforehand we were able to achieve better quality then we would have, if we just came up with it while implementing.

6   Conclusions

6.1   Achievement and Project Goal

In Order to talk about our Achievements in this project and whether or not our Project Goal was met, we first need to specify what our Project Goal was.

The Goal of this project and the purpose of the Software was to improve the Usability and fasten the tasks of managing the SWT library, so that the Chairmembers and staff can dedicate more time into research and teaching, while students can work with an accessible, easy to use and reliable software in order to find the books they are interested in. Furthermore the Program offers a better overview of who lend which book, it eases up that process compared to the currently existing one.

During this Project we also achieved some minor but also some major things.

The Greatest thing that we as a team achieved, is that we picked up a technology that we were not familiar with, expanded our knowledge about it and Managed to write a Reliable software. Some of the more minor achievements is the fact that we managed to consistently meet at least twice a week, we stayed in touch via electronically channels, and were always ready to help each other. We came from a group of Strangers and became friends during the process.

6.2   Future Work

SWTlib is full of improvement and room to grow. There are several more features which can be added in to improve the experience even more for both, students and Chairmembers.

Some of these features could be a Comment section in each Detailed book View, so users can share their opinion how helpful a specific book was for them in more detail, rather then just up- or downvote.

Furthermore there could be an Image added to each book, so that the user does not only see the title but also the cover of it, as this is often a way of recognize a specific book which they might not know the title of.

The UI could be improved in a way that it satisfy the User while using the software.

At a Further point in time videos of Lectures could be uploaded there as well, if a student missed a lecture where he absolutely could not come.

SWTlib has a lot of potential and therefor the possibilities to expand on it are limitless.

A   Log of Working Hours

A.1   Simon Nützel

Year/Week	Hours	Achievement
2019/ W20	8	Gather and document requirements by various means such as studying the project brief or analyzing similar software
2019/$\sim W20$	5	Come up with and document several use-cases that would end up being demonstrated with our presentation prototype
2019/$\sim W20$	2	Draw a system context diagram.
2019/$\sim W20$	4	Come up with ideas for useful statistics and graphs that might be interesting to visualize
2019/$\sim W20$	34	Study ASP.NET, C# in preparation for working with these technologies
2019/$\sim W23$	15	Set up and explore mandatory software such as Visual Studio and Docker
2019/$\sim W23$	18	Research authorization tech
2019/$\sim W25$	35	Implement log in system
2019/$\sim W25$	7	Work with the database in MySql Workbench, e.g. populate it with sample data or manipulate values for testing purposes
2019/$\sim W25$	14	Work on roles system
2019/$\sim W25$	20	Pair programming and group sessions
2019/$\sim W28$	6	Research library data API for the potential auto complete feature
2019/$\sim W30$	12	Work on the project report.
Total Hours:	180	

A.2 Daniel Rooney

Year/Week	Hours	Achievement
2019/ W20	4	Analyze project brief and study the requirements and scope of the project for preparation.
2019/$\sim W20$	3	Design and construct wire-frame diagrams that would prove beneficial to our presentation prototype. Focus on the design of the User Interface with functionality in mind.
2019/$\sim W20$	5	Develop existing wire-frame diagrams and design ideas, with a focus on simplicity and accessibility.
2019/$\sim W20$	7	Research and compile useful and beneficial additional functionality ideas that could be implemented into the final product.
2019/$\sim W20$	25	Research, practice and familiarize one self with the designated functionality and technologies to be used in the project. (ASP.NET Core MVC, Entity framework, mySQL etc.)
2019/$\sim W23$	21	Begin initial work on code skeleton and familiarize one self with the IDE and frameworks (VS2019, MVC model etc.)
2019/$\sim W23$	13	Implement category functionality to dashboard.
2019/$\sim W25$	27	Implement basic search functionality in conjunction with other team member.
2019/$\sim W25$	5	Design, implement and proof read cookie policy and privacy page.
2019/$\sim W25$	9	Implement statistics page using chart.js library.
2019/$\sim W25$	20	Rapid development with other members of the team. Integration of statistics functionality using mySQL workbench.
2019/$\sim W28$	8	Completion of statistics page functionality and UI design improvements.
2019/$\sim W30$	18	Document work and transfer all documentation to the project report. Complete designated sections of project report (Problem analysis and organization).
Total Hours:	165	

A.3   Sergej Gelver

Year/Week	Hours	Achievement
2019/ W20	*6*	*Analyze Project brief, think of Requirements, think of scope of the project*
2019/ $\sim W20$	*3*	*Think of a UI and create a PowerPoint Prototype*
2019/ $\sim W20$	*4*	*Team meetings for Architecture decisions*
2019/ $\sim w20$	*30*	*Inform myself about the Technology used (ASP.net core/Entity Framework/C)*
2019/ $\sim W20$	*6*	*Get to know about other systems used (Docker/Visual Studio)*
2019/ $\sim W21$	*1*	*Gather ideas for useful Statistics*
2019/ $\sim W25$	*10*	*Supported the team in various areas*
2019/ $\sim W27$	*20*	*Basic click Tests to check for bugs/unintented behaviour*
2019/ $\sim W28$	*20*	*Work on Reminders functionality*
2019/ $\sim W28$	*4*	*preparing AraCom presentation*
2019/ $\sim W29$	*12*	*More Tests*
2019/ $\sim W30$	*25*	*Work on Project report*
Total Hours:	*141*	

A.4   Zdenek Scherrer

Year/Week	Hours	Achievement
2019/ W20	2	Get familiar with project brief, gather functional and non-functional requirements
2019/ $\sim W20$	3	Refactor the PowerPoint prototype and add additional functionalities
2019/ $\sim W20$	4	Team meetings for Architecture decisions
2019/ $\sim W20$	20	Get myself familiar with Databases, Entity Framework, ASP.NET Core
2019/ $\sim W20$	2	Create ASP.NET Core MVC code skeleton
2019/ $\sim W21$	1	Added Entity Framework Core to the code
2019/ $\sim W22$	4	Added Docker support and prepared the CI/CD pipeline
2019/ $\sim W23$	4	Added basic layout and tried to fix docker support
2019/ $\sim W24-25$	14	Added Rental View, Rental Logic, Books View, Book Detail View, Book Logic, Author Logic, Database connection
2019/ $\sim W27$	35	Merged a lot of work regarding rentals, books, bookmarks, style, database
2019/ $\sim W28$	20	Several changes to Views, Rental System, Book Management, Style, Database
2019/ $\sim W28$	10	Tried to fix CI/CD pipeline
2019/ $\sim W28$	16	Added Right navigation items, view components, reminders, basic xml-upload
2019/ $\sim W29$ advanced search	12	Fixed some content bugs, implemented search functionality
2019/ $\sim W29$	26	Reminders fix, Rentals fix. Added Wishlist. Rework everything to take real UserId. Fix several Bugs
2019/ $\sim W30$	25	Work on Project report "Design". Still fixing some bugs.
Total Hours:	198	

Ehrenwörtliche Erklärung

Alle Unterzeichner erklären hiermit, dass sie die vorliegende Arbeit (bestehend aus dem Projektbericht sowie den separat abgelieferten digitalen Werkbestandteilen) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

	Full name of Student 1
Matrikelnummer	*Name*
Ort, Datum	*Unterschrift*

	Full name of Student 2 (if applicable)
Matrikelnummer	*Name*
Ort, Datum	*Unterschrift*

	Full name of Student 3 (if applicable)
Matrikelnummer	*Name*
Ort, Datum	*Unterschrift*

	Full name of Student 4 (if applicable)
Matrikelnummer	*Name*
Ort, Datum	*Unterschrift*

	Full name of Student 5 (if applicable)
Matrikelnummer	*Name*
Ort, Datum	*Unterschrift*