Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

József Urak

FULL STACK CEMETERY MANAGEMENT SYSTEM WITH INTERACTIVE MAPS USING REACT AND ASP.NET CORE

SUPERVISOR

Zoltán Benedek

BUDAPEST, 2025

Table of contents

Abstract **5**
1 Introduction **6**
1.1 Choice of topic 6
1.2 Existing solutions 6
1.2.1 Interactive Maps 7
1.2.2 Memorial Features 7
1.2.3 Integrated Blogs 7
2 Technologies used **8**
2.1 .NET 8
2.2 ASP.NET Core 9
2.3 Entity Framework Core 9
2.4 ASP.NET Core Identity 10
2.5 TypeScript 11
2.6 React 12
2.7 Chakra UI 13
2.8 Architectural and Design Patterns 13
2.8.1 Clean Architecture 13
2.8.2 Repository pattern 15
3 Requirements **16**
3.1 Roles 16
3.2 Functions related to the data of deceased persons 16
3.3 Memorial page features 17
3.4 Grave-related functions 17
3.5 Interactive map 18
3.6 Non-functional requirements 18
4 Architectures **19**
4.1 System architecture 19
4.2 Application server architecture 20
4.2.1 Business logic layer 21
4.2.2 Data access layer 22
4.2.3 Web API layer 22

4.2.4 Summary 23
4.3 Database design 24
4.3.1 AspNetUsers 25
4.3.2 AspNetUserRoles 25
4.3.3 AspNetRoles 25
4.3.4 Grave 26
4.3.5 Deceased 26
4.3.6 Message 26
4.3.7 Polygon 26
4.3.8 Point 26
4.4 Client application architecture 27
5 Implementation 29
5.1 Styling 29
5.2 User management 29
5.2.1 Login 30
5.2.2 Registration 33
5.2.3 Authorisation control 35
5.3 Search page 35
5.4 Memorial page 37
5.5 Map display 38
5.5.1 React Leaflet 38
5.5.2 Custom map 39
5.5.3 Tile generation 40
5.5.4 Map display 41
5.5.5 Displaying graves on the map 43
5.5.6 Cemetery caretaker functions 45
6 Evaluation 49
6.1 Experience 49
6.2 Opportunities for further development 49
7 References 51

STUDENT STATEMENT

I**, József Urak**, a postgraduate student, hereby declare that I have authored this thesis without any unauthorised aid, using only the sources specified (literature, tools, etc.). I have clearly indicated any sections that I have taken verbatim or in the same meaning but rephrased from other sources by citing the source.

I agree that the basic data of my work (author, title, English and Hungarian abstract, year of completion, name(s) of supervisor(s)) may be published by BME VIK in a publicly accessible electronic form, and that the full text of the work may be published on the university's internal network (or for authenticated users). I declare that the submitted work and its electronic version are identical. In the case of thesis plans classified as confidential with the dean's permission, the text of the thesis will only become accessible after 3 years.

Dated: Budapest, 6 December 2024

...…….

József Urak

Abstract

Modern web applications have opened many new possibilities in the field of record-keeping and community internet platforms. These applications not only provide data management but also make them interactive and easily accessible.

Cemetery registry websites have become widespread in Hungary. These platforms often serve not only as record-keeping tools but also as community platforms. Such systems allow people to share their memories.

These applications offer diverse and convenient features for users. For example, individuals responsible for cemetery management can easily manage grave sites and deceased records. This information becomes accessible to anyone from the comfort of their home. Additionally, users could share their own memories and even, for example, light virtual candles in memory of their loved ones. In conclusion, these websites not only reduce administrative burdens associated with cemeteries but also help preserve memories.

During the development of the application, I employed a multi-layered architecture. In addition to the traditional three layers, I created additional layers. The web server is based on ASP.NET Core, while the client-side application is built on the React framework. A REST API facilitates communication between the web server and the client. The goal was to create a system that is easy to maintain and extend. During the design process, I focused on creating independent components. I also familiarized myself with technologies such as React and TypeScript, which I describe in more detail in my thesis.

1 Introduction

1.1 Choice of topic

Most cemeteries have kept records in some form for centuries, as it is important to organise and keep track of grave sites and data on the deceased to prevent the memory of our ancestors from fading into oblivion. Traditionally, this work was conducted by the cemetery caretaker, with the help of clergy. However, in the age of digitalisation, this work has now been transferred to the internet in many places with the help of websites.

Online platforms allow cemetery administrators to store data in a central location and keep it up to date. In addition, relatives and interested parties can now search online and retrieve information about cemeteries and those buried there from the comfort of their own homes.

Thanks to the possibilities offered by websites, the memory of the deceased is easily accessible not only during on-site visits, but anytime and anywhere. Incidentally, it also helps people to easily find a grave they are looking for when visiting a cemetery, even if they do not know its exact location, thus saving them a lot of searching and inconvenience.

In addition, online cemetery registers can also serve as community platforms, providing an opportunity for people to remember their loved ones together.

Modern internet applications offer significant development opportunities for such cemetery register and record-keeping systems. These applications make it possible to create an interactive map of the cemetery and implement features such as blog-like messages, memorial pages and many other features that can be found on the latest cemetery register websites today.

1.2 Existing solutions

During the modernisation of cemetery cadastre and registration systems, several innovative solutions have been put on the table. These innovations not only facilitate the administration of cemeteries but also enable the preservation of memory and make remembrance more interactive, using several modern features. I will present some of them in detail below.

1.2.1 Interactive Maps

Interactive maps allow for virtual viewing of cemeteries. Users can easily navigate between graves, view information about the deceased, and locate important objects in the cemetery, such as funeral parlours or memorials. These maps help those involved to find the graves they are looking for quickly and easily.

1.2.2 Memorial Features

The numerous memorial features offer relatives and friends the opportunity to share their memories and pay their respects to the deceased. These platforms allow users to write comments or even place virtual memorial items in memory of the deceased, such as candles, wreaths, stones, flowers, depending on their religion. This makes the memory of the deceased not only personal but also communal, and this approach contributes to the cultivation and preservation of memories.

In addition, some similar websites offer several services related to funerals and grave maintenance, which further enhance the functionality and usefulness of the applications.

1.2.3 Integrated Blogs

Integrated blogs allow cemetery administrators and stakeholders to share up-to-date information about cemetery events and news. These blogs help people stay up to date on changes and events related to the cemetery.

2 Technologies used

2.1 .NET

.NET is a developer platform released by Microsoft, the first version of which was released in 2016, then under the name .NET Core. When .NET five was released, the Core designation was dropped, emphasising that this is in fact the main implementation of .NET. When I started developing my application, the latest version released was .NET eight. As I discovered during my work, many of its new features were useful to me [1] .

.NET is free and open source, maintained jointly by Microsoft and the community. The source code for .NET is licensed under the MIT licence. New versions of .NET are usually released in November each year[2] .

During its development, the main goal was to make it modular, fast, lightweight, but also secure and dependable, unlike its predecessor, the .NET Framework. This was achieved by including only basic features. Previously, this was also referred to as "Core". Additional functionality is provided by so-called NuGet packages, which can be added to our application as needed.

The .NET platform is platform-independent, so it can be used not only on Windows systems, but also on Linux and macOS. It also supports several programming languages, the most used being C#, but F# and Visual Basic are also used to develop .NET applications. C# is a strongly typed, object-oriented language. Most of .NET is also written in C#. To use C#, you need the "GC" (Garbage Collector), which is a tracing garbage collector that provides automatic memory management. This makes .NET more secure and dependable, as it eliminates many vulnerabilities.

With .NET, we can develop applications for several platforms. The application models are based on the base libraries.

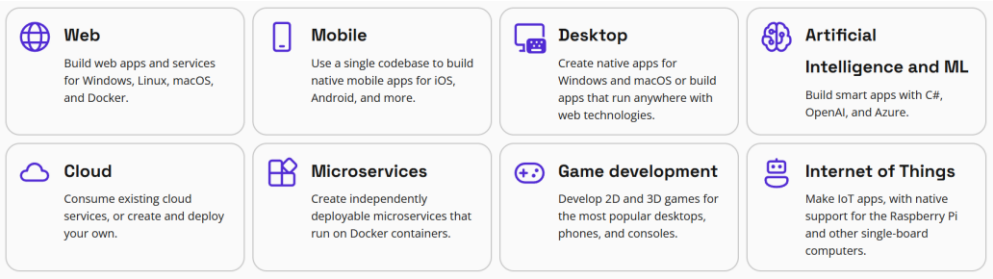

Figure 1: Overview of platforms and application models supported by .NET [1]

2.2 ASP.NET Core

ASP.NET Core is an efficient, open-source web framework developed by Microsoft. It enables the rapid and efficient development of simple websites, APIs, or complex web applications. Its modular structure allows you to select unnecessary features, which minimises the size of the application and increases its performance.

ASP.NET Core is platform independent. This makes applications widely accessible and easily scalable. .NET-based ASP.NET applications can also be deployed and maintained in Docker containers, which is beneficial during development and operation. Overall, ASP.NET Core is a modern, efficient, and versatile framework for developing any type of web application.

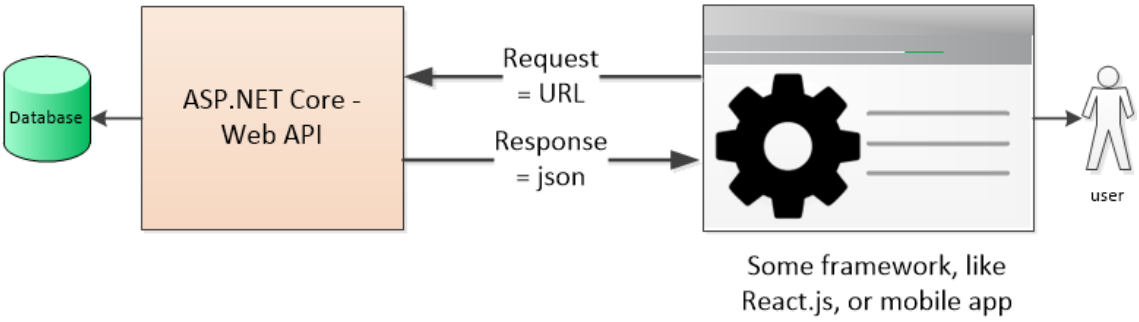

Figure 2: How ASP.NET Core Web APIs work [3]

2.3 Entity Framework Core

Entity Framework (EF) Core [4] is a modern data access technology developed by Microsoft that helps .NET developers work with databases easily and efficiently. EF Core is a lightweight, extensible, open-source, and platform-independent framework that was released in 2016 as an enhanced version of the original Entity Framework. EF Core supports the most popular database engines, such as SQL Server, PostgreSQL, MySQL, and SQLite databases.

The main goal of EF Core is to facilitate working with databases while reducing the amount of code required and making the development process more efficient. EF Core allows database tables to be represented as .NET objects and queried using LINQ (Language Integrated Query). This allows developers to work directly with .NET language elements on databases, simplifying their work.

In EF Core, data access is done through a model. This consists of entity classes and a context object (DbContext). Entity classes represent database tables and their columns, while the context object provides communication with the database. Using the context, we can perform queries and save data. The model can be created either by generating it from an existing database (Database First) or by coding it manually (Code First).

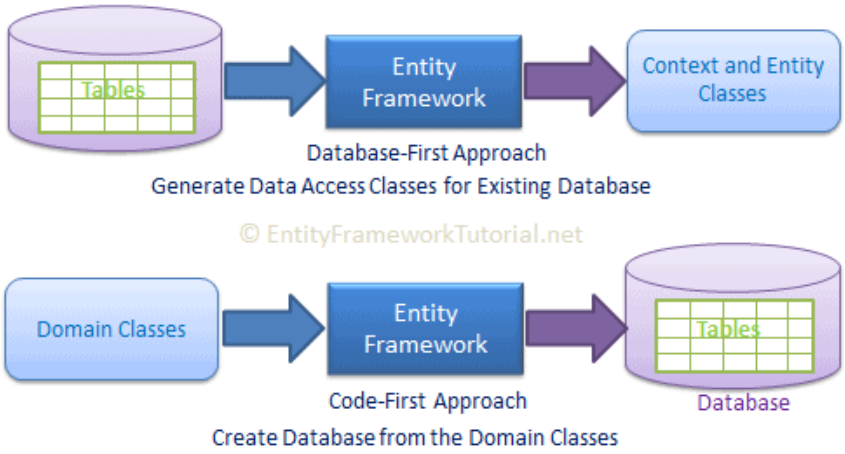

Figure 3: Database-First and Code-First approaches in Entity Framework. [5]

One useful feature of the tool is database migration management. Migrations allow us to easily transfer model changes to the database, keeping it up to date with the application code base. Using migrations, database structure maintenance can be automated, resulting in considerable time and labour savings.

2.4 ASP.NET Core Identity

ASP.NET Core Identity[6] is an API that supports the login functions of the user interface of ASP.NET Core web applications. This API allows you to manage users, including passwords, profile data, roles, permissions, tokens, and email confirmations. Users can create their own accounts, which are stored by Identity, or they can use external login providers such as Facebook, Google, Microsoft Account, or Twitter.

Identity is typically configured when the application is launched, where, in addition to adding and configuring services, authentication and authorisation middleware components are also set up in the request pipeline. When using Identity, registration, login, and logout operations can be easily integrated into our application. Built-in templates allow us to quickly create these functions, which contain the necessary

endpoints, such as /Identity/Account/Login, /Identity/Account/Logout, and /Identity/Account/Manage.

ASP.NET Core Identity also supports two-factor authentication (2FA), which increases security by requiring a second authentication step from users when they log in. This could be a code sent via SMS or a time-based one-time password (TOTP) generated by a mobile application.

Thanks to the flexibility of the Identity system, it is customisable, allowing developers to easily add custom user fields, rules, or even new authentication services. The Identity database schema uses Entity Framework Core by default, but other database management systems can also be integrated.

Another important feature of ASP.NET Core Identity is data protection and data security. To ensure secure password storage, the system uses strong hash algorithms and supports data protection standards such as GDPR. The Identity system provides the option to encrypt data and use tokens to meet data protection requirements.

2.5 TypeScript

JavaScript first became popular as a client-side language, but with the advent of Node.js, it also became widespread on the server side. TypeScript is not just a new language, but an entire ecosystem that contributes significantly to the development of JavaScript-based applications[7] .

TypeScript is a strongly typed, object-oriented programming language developed by Anders Hejlsberg, the designer of the C# language, at Microsoft. TypeScript is not only a programming language, but also a set of tools. TypeScript is an extended version of JavaScript, enhanced with new features, and converted into JavaScript code.

ECMAScript is a standard for scripting languages, with six editions published as ECMA-262. The sixth edition, called "Harmony", was followed by the release of TypeScript, which is consistent with the ECMAScript 6 specification.

TypeScript borrows its basic language elements from the ECMAScript 5 specification, which is the official specification for JavaScript. TypeScript language elements such as modules and components also comply with the ECMAScript 6 specification. In addition, TypeScript includes built-in generic types that are not included in the ECMAScript 6 specification.

One of the advantages of TypeScript is that it allows code errors to be detected at compile time, before the script itself runs. In addition, TypeScript offers strong static typing, which helps to avoid errors when writing code. The language supports object-oriented programming principles such as classes, interfaces, and inheritance, as well as the use of type definitions from existing JavaScript libraries. These advantages help developers work more efficiently and in a more structured manner on their projects.

2.6 React

React is a JavaScript-based open-source library designed primarily for developing user interfaces. It is one of the most popular choices among front-end developers. It was first released in May 2013 and was developed by Facebook[8] .

Reacts main principles are simplicity and efficiency. Developers break their applications down into separate pieces, called components, which perform different tasks. This allows the application structure to be clean, reusable, and easy to maintain.

The Virtual Document Object Model (Virtual DOM) technology used by React treats the user interface as a tree structure and allows the application to respond efficiently to user interactions, minimising performance losses. This is because, unlike many older technologies, it does not rebuild the entire interface when the content of the web page changes but compares the new state with the previous one and only updates what has changed.

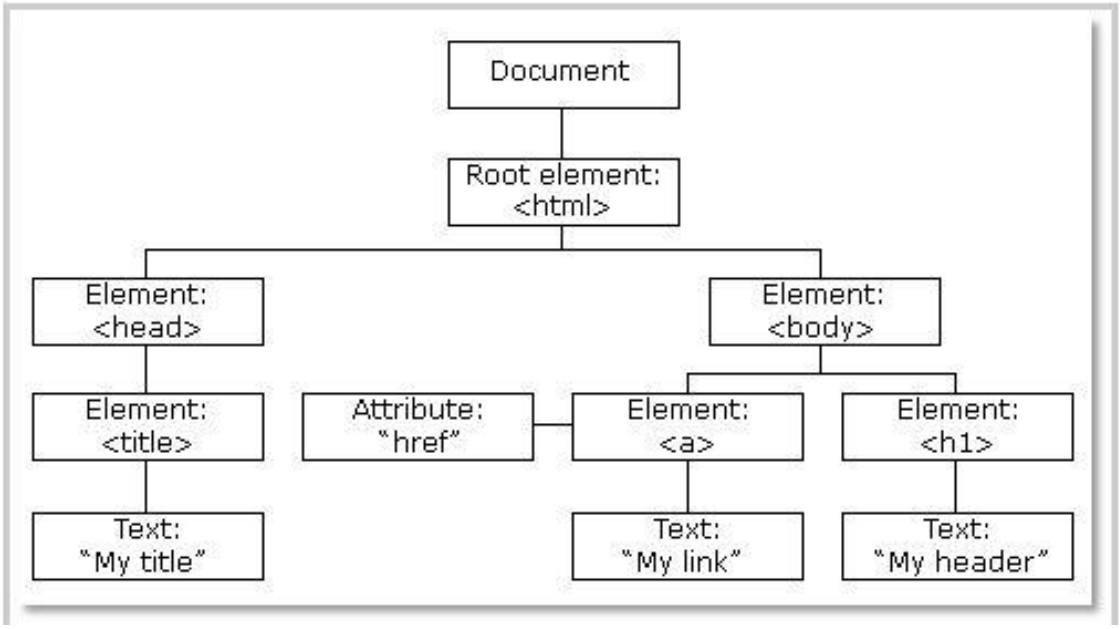

Figure 4: Virtual Document Object Model (Virtual DOM) example [9]

The advantages of React include fast development and ease of learning, which allow developers to efficiently create complex user interfaces. In addition, widely available documentation and a large community help developers solve problems and improve their applications.

React is suitable for developing user interfaces for both web and mobile applications. One branch of React, the React Native framework, is used to create the appearance of mobile applications. Thus, React can be used effectively in both worlds, so developers do not need to learn separate technologies for developing web and mobile applications.

2.7 Chakra UI

Chakra UI is a modern, user-friendly React component library developed by Segun Adebayo, who designed Chakra UI for quick and easy creation of user interfaces. A wide selection of pre-designed and highly customisable components allows developers to easily and efficiently create complex user interfaces[10] .

Chakra UI offers React components that developers can easily integrate into any application. These components cover a wide range of user interface elements used in modern applications, such as buttons, forms, modals, and more. One of the notable features of Chakra UI is its emphasis on styling props, which allow developers to style components using props.

Chakra UI's easy-to-use and customisable components, as well as its intuitive and beginner-friendly API, allow developers to quickly understand and use the components. In addition, Chakra UI supports the so-called tree-shaking technique, which selectively removes dead code during build processes, significantly reducing the final package size.

Furthermore, it employs a modular structure that allows for the isolation of core libraries and the selective import of only those components that are needed for rendering.

2.8 Architectural and Design Patterns

2.8.1 Clean Architecture

Clean Architecture, popularised by Robert C. Martin, is a software design approach that aims to ensure the cleanliness and maintainability of system structures [11] . The basic principle is to divide the software system into several well-separated layers,

each with clearly defined responsibilities. This approach allows systems to be flexible, easy to test and maintain.

Clean Architecture differs fundamentally from traditional three-layer architecture in that it strictly separates layers and minimises their dependencies. As a result, changes in the data access layer, for example, do not affect the domain or application layers, ensuring greater maintainability of the system, but also the reusability of individual layers.

The most important principle of clean architecture concerns the direction of dependencies between layers. More specifically, internal layers do not depend on external layers. This means that the business logic layer (Domain) is located at the centre and does not depend on any external layers, such as infrastructure, user interface, or any external systems.

The next layer is the Application layer, which communicates with the Domain layer and uses its business logic to perform specific tasks. This layer contains use cases that define how the system should behave in response to different user interactions. The application layer remains independent of the outer layers.

The infrastructure layer contains all solutions responsible for connecting the application layer and the outside world. This includes, for example, database access layers, external API integrations, and other technical details. The infrastructure layer acts as a bridge, ensuring that the application layer runs smoothly in the real world.

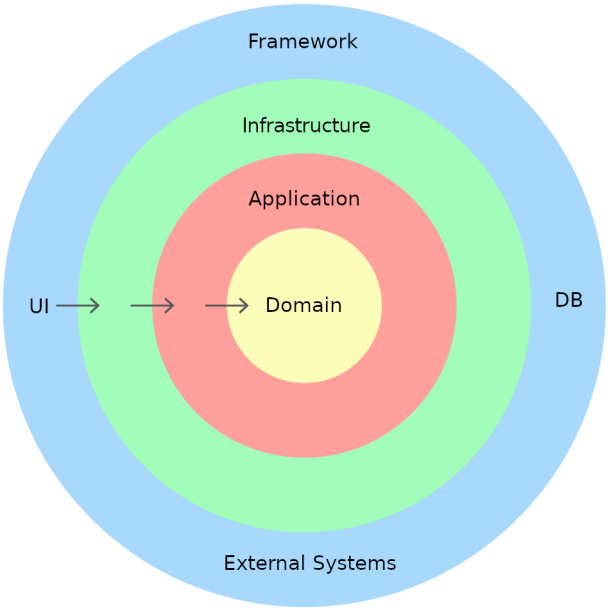

Figure 5: Layers of Clean Architecture and their dependencies [12]

2.8.2 Repository pattern

The Repository pattern is a design pattern often used to structure the data management layer of applications. Its purpose is to separate the data access layer from the rest of the application, making it easier to maintain and test the code [13] .

This pattern usually consists of classes or interfaces that are responsible for storing, querying, and modifying data. Repositories function as intermediaries between the application's business logic and the data access layer, so that the business logic has no direct connection to the database or other data sources.

Repository interfaces are used throughout the application to access data without knowing the exact details of how data access works. This makes the application code cleaner and easier to maintain, as data access operations are completely separated from business logic.

One of the advantages of the Repository pattern is improved testability. Since repositories communicate with business logic through interfaces, it is easy to inject dependencies and use mock objects during unit testing. This allows us to simulate database operations during testing without using a real database, which makes the testing process faster and more dependable. In addition, the Repository pattern also supports data source interchangeability, which can be particularly useful if the application needs to switch to a different database or other data storage solution later, as the changes only affect the Repository layer and not the business logic.

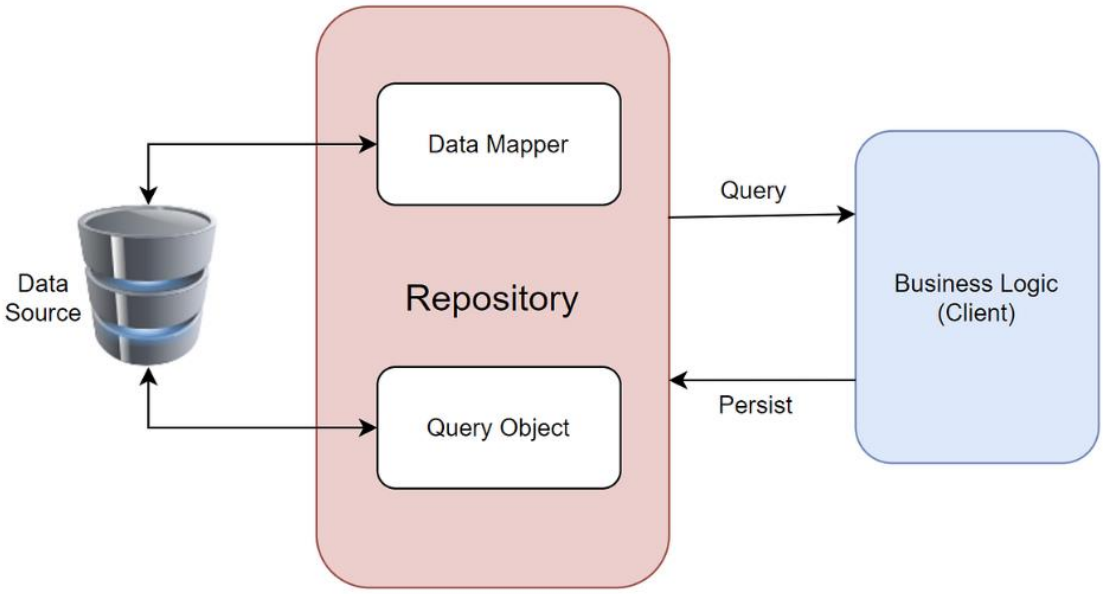

Figure 6: Illustration of how the Repository pattern works [14]

3 Requirements

In this chapter, I will describe both the functional and non-functional requirements for the application. My goal is to present in detail the criteria that had to be met for the application to function properly.

In the following chapters, I will illustrate the functions using a use case model, separating the functionality available to distinct roles.

3.1 Roles

As in any system designed for record-keeping purposes, it is essential to define distinct roles in the Cemetery Register web application. The application distinguishes between two main roles: the administrator and the simple user. Certain functions are only available to the administrator, ensuring the secure management of the system's data.

3.2 Functions related to the data of deceased persons

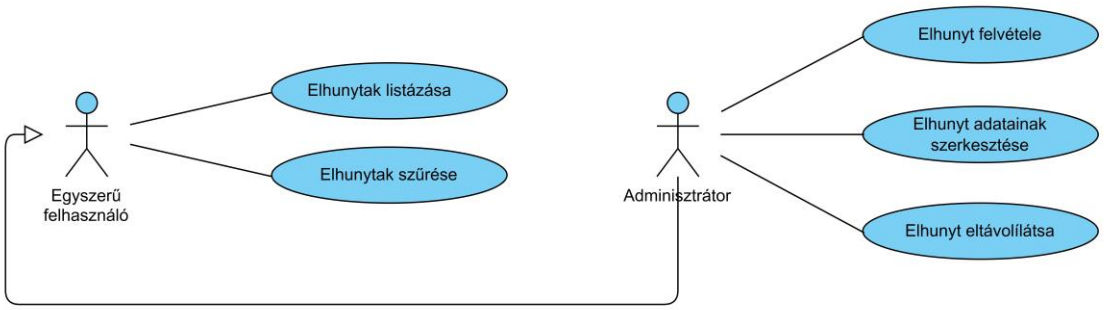

Figure 7: Functions related to the data of deceased persons

Regular users have basic permissions: they can access the list of deceased persons and search and filter among them. This allows all users to find the basic data of any person buried in the cemetery.

The administrator has higher-level permissions: they can not only browse the data but also add new deceased persons to the system and modify existing data.

3.3 Memorial page features

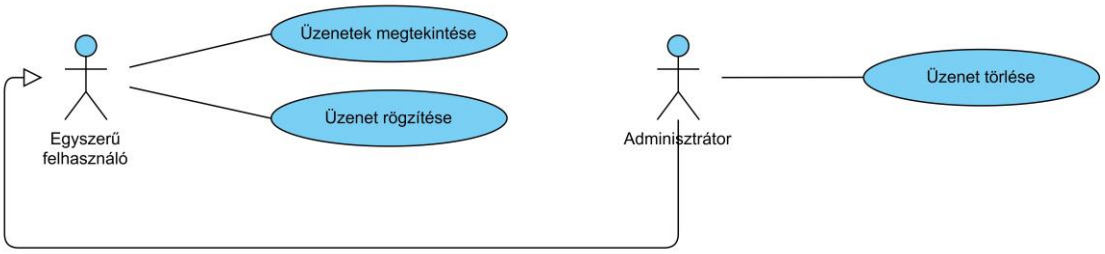

Figure 8: Memorial page features

Regular users have access to two basic functions: viewing messages and posting new messages. These functions allow relatives and other mourners to leave messages in memory of the deceased and read the tributes of others. This interface thus also serves as a community memorial platform where users can share memories and express their respect.

The administrator role has higher privileges. The message deletion function allows the administrator to remove any inappropriate or offensive messages that do not fit the purpose or ethics of the site.

3.4 Grave-related functions

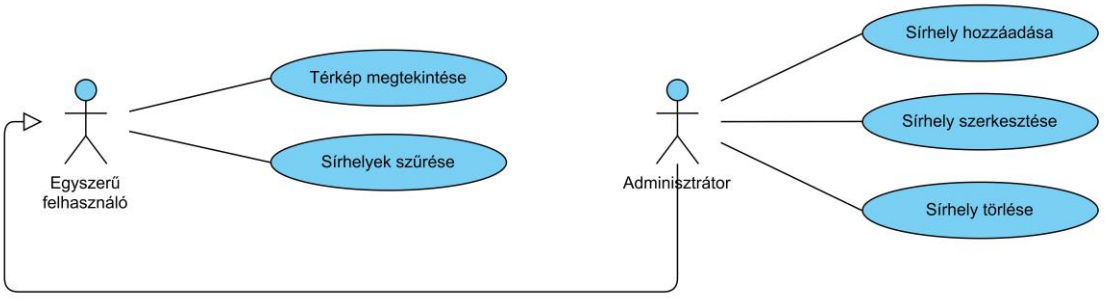

Figure 9: Grave-related functions

Viewing the cemetery map and filtering graves is freely available to all users. These functions allow users to easily browse the cemetery area and quickly find the graves they are looking for using the filter options. This is particularly useful for visitors, as it allows them to view the cemetery and quickly identify the location of the graves they are looking for.

The administrator role has significantly expanded permissions, allowing them to designate new graves, as well as edit and delete them. The editing function includes modifying the coordinates of the polygon marking the grave and assigning the deceased to the appropriate graves.

3.5 Interactive map

The purpose of the cemetery map is to provide interactive navigation for users. It is important that the map allows users to zoom in on different areas of the cemetery and view individual graves in more detail. Graves must also be marked graphically on the map as polygons.

The map also includes a search and filter function for graves. This allows users to quickly find the graves they are looking for based on name, year of birth or death, or other parameters. Based on the search results, they can easily identify the grave they are looking for on the map.

Another important feature is the ability to view information about the deceased: by clicking on a grave, users can access information about the deceased buried there, such as their name, date of birth and date of death.

3.6 Non-functional requirements

The non-functional requirements of the application include ensuring fast and smooth operation, especially for the search and filter functions. Pages and functions should load with minimal waiting time so that users can navigate the application easily and smoothly. In addition, the structure of the application should be simple and clean.

Security and data protection are also important considerations. Access to the data of deceased persons should be controlled so that only administrators can perform data management operations.

4 Architectures

In this chapter, I will present the basic structure of the application I have created. First, I will look at the application, then I will describe the role, task, and structure of each component in more detail. When designing the application, I strove to create as weak a link as possible between the different layers, keeping in mind their reusability, testability, and maintainability.

4.1 System architecture

The application I created consists of three main components. This architecture logically and physically separates the functional parts of the application, ensuring maintainability, scalability, and easier further development. Communication between components takes place through clearly defined interfaces.

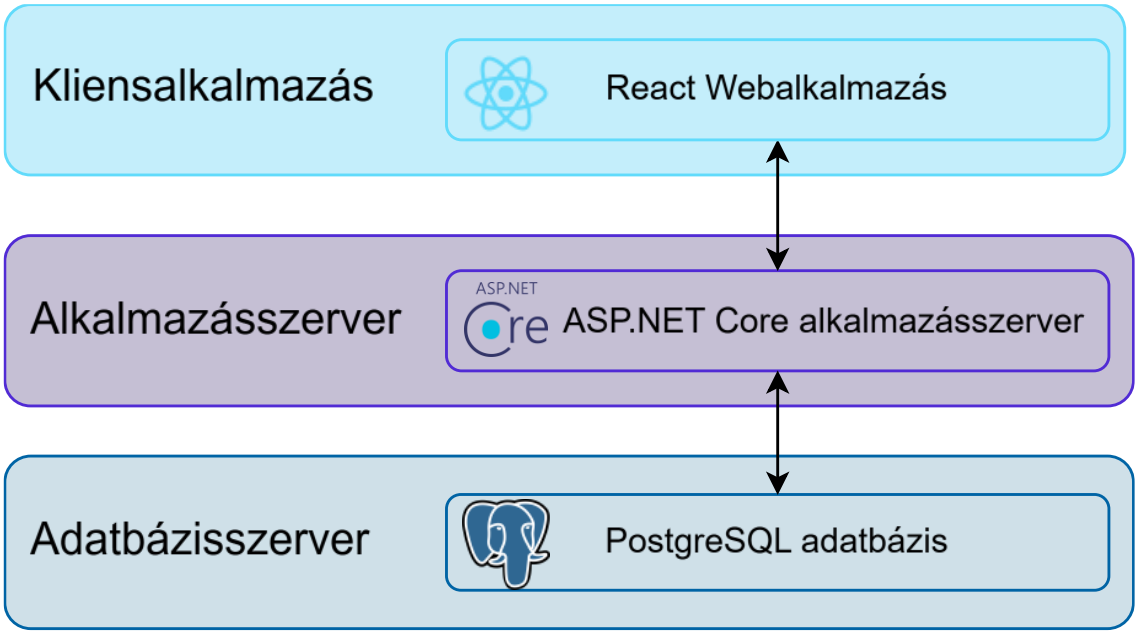

Figure 10: System architecture

The first component is the client application, which is based on the React library and corresponds to the presentation layer. This layer is responsible for handling user interactions and displaying data visually. Communication with the application server takes place via REST API.

The second component is the application server. This server is responsible for processing user requests, validating data, applying business rules, and preparing data for

the client. The application server works closely with the database server, and Entity Framework maps relational database data to objects for the application.

The third component is the database server, which uses the PostgreSQL database management system. This layer is responsible for persistent storage of data, handling queries based on the relational data model, and maintaining data integrity. The database server ensures fast and reliable access to data for the application server.

These three components work closely together to ensure the full functionality of the application, while performing strictly separated functions. This structure allows the application to be modular, easy to maintain, and technologically flexible.

4.2 Application server architecture

The project structure shown in Figure 11 follows the structure and basic principles of Clean Architecture, also known as Onion Architecture. This approach, in line with the fundamental goals of Domain-Driven Design, ensures that business logic functions as the central element of the application, completely independent of the outer layers, especially the implementation of the infrastructure layer[15] .

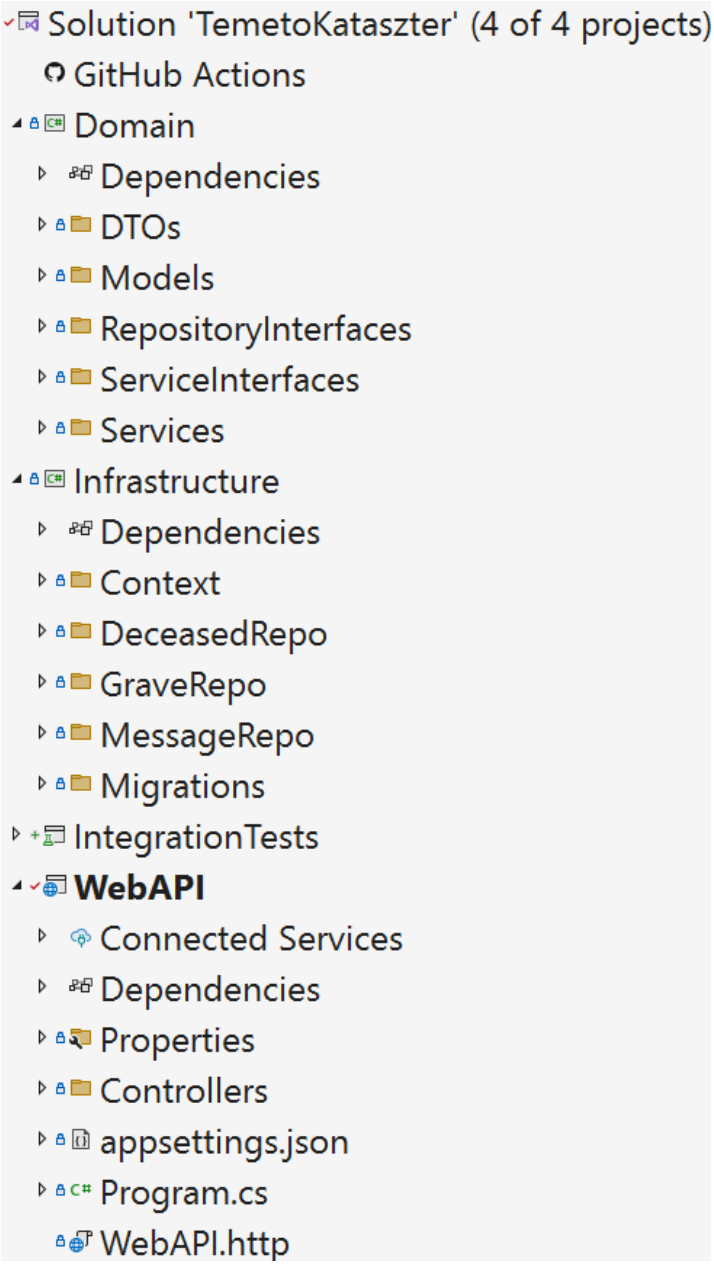

Figure 11: Application server structure

4.2.1 Business logic layer

The business logic layer, or domain layer, is the central element of the application, representing the innermost layer of the Clean Architecture model. This layer contains the business logic and the basic concepts necessary for the application to function, such as business entities, which describe the most important data structures of the application and their relationships (Models).

In addition, this layer also contains data structures called Data Transfer Objects (DTOs)[16] . These objects facilitate data communication between layers, particularly in the transmission and reception of data to and from external layers. The purpose of DTOs

is to transfer only the necessary data during communication, while avoiding the direct transfer of business entities.

The Business Logic layer also defines interfaces that specify the access points for data and business services. The use of interfaces ensures that the business logic is independent of specific implementations.

Services that operate based on the interfaces and execute the specified logical processes implement business rules within this layer. This layer is responsible for the complete implementation of the application's business rules, ensuring that they remain independent of technological details or infrastructure.

4.2.2 Data access layer

The data access or infrastructure layer is responsible for the specific implementation of data access. This layer implements the repository interfaces defined by the business logic, providing the functionality necessary for data management.

Entity Framework plays a significant role in this layer. The Context folder contains the database connection and object-relational mapping (ORM) configurations, which enable automatic mapping between relational database structures and entities.

The various Repository folders, such as DeceasedRepo, GraveRepo, or MessageRepo, implement specific solutions required for individual data storage operations. These implementations include operations such as querying, modifying, deleting, or adding data.

The Migrations folder is used to manage database migrations, which ensure that the database structure is up to date and allow model changes to be synchronised with the database.

4.2.3 Web API layer

The Web API layer implements the external interface of the application and is responsible for communicating with the client-side application. This layer implements ASP.NET Core-based RESTful APIs, which are managed by controllers located in the Controllers folder. These controllers receive HTTP requests, forward the requests to business logic services, and then return the results to the client side.

The Web API layer configurations are in the appsettings.json file, which contains, among other things, the database connection parameters. The entry point of the application is the Program.cs file, which is responsible for initialising the application. This file configures dependency injection (DI), which enables cooperation between different layers, resolving numerous dependencies.

In addition, this is where you register and load additional middleware layers. Examples include Identity (user identification system) and Authorisation (access control) provided by ASP.NET Core.

4.2.4 Summary

The figure below illustrates the cooperation between the three components using a specific example, which shows the addition of a deceased person's data. This example serves as a model for the operation of other system operations, as they are all performed in the same way. The figure makes it easy to understand the relationships and processes between the components.

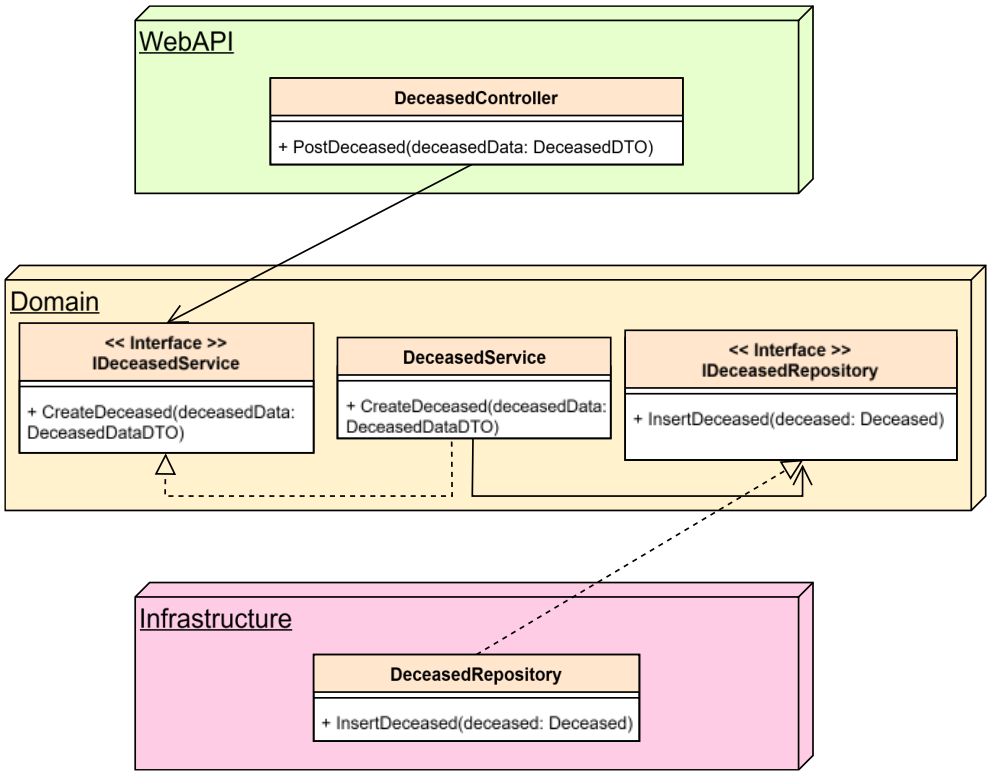

Figure 12: Example of the process of adding a deceased person

In the Web API layer, the controller corresponding to the given domain entity receives the client's HTTP request in the function corresponding to the currently called

endpoint. In the example examined, this is the PostDeceased function, which manages post-type requests related to the Deceased entity.

This function typically does nothing more than forward the incoming operation and data to the appropriate service in the Domain layer. This type of functionality typically follows the so-called "Thin Controller" principle, which requires the controller layer to operate in a minimalist manner and not contain any business logic [17]. It is important to note that the application still transmits data in DTO form.

In the Domain layer, the given service performs business logic processing, which includes, for example, DTO mapping. The service transfers the processed data to the infrastructure layer. The Repository interface (IRepository) provides the appropriate abstraction and separation from the specific Repository implementation, which performs the actual database operations.

In the Infrastructure component, the Repository uses an appropriate method to save the entity data that the service previously converted and processed from DTO to the database.

4.3 Database design

The database uses a PostgreSQL relational database and applies Entity Framework for object-relational mapping. The system uses a code-first approach, which allows developers to create the database structure by defining model classes, and then Entity Framework automatically generates the appropriate tables and relationships. Identity automatically creates seven tables containing user data, but I have only included those that I use in the project in the diagram below.

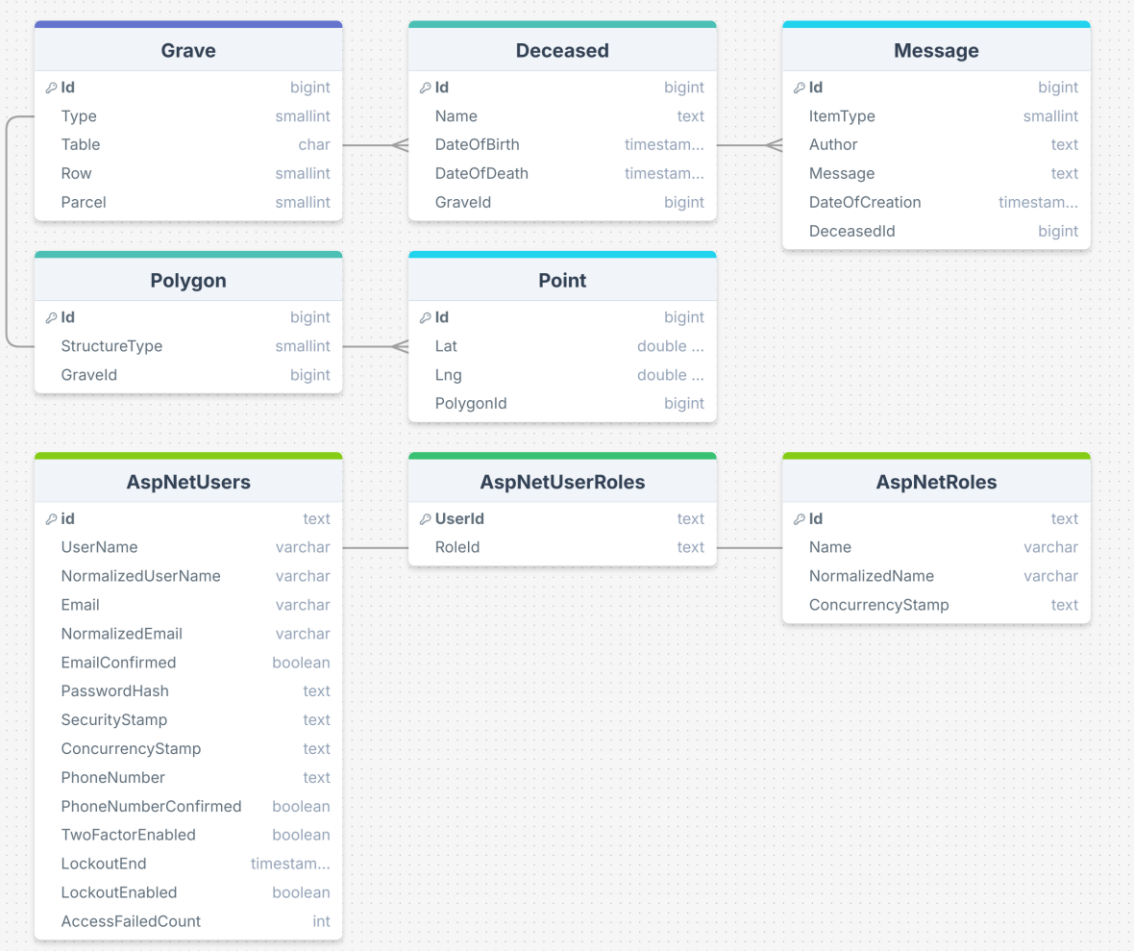

Figure 13: Database schema

4.3.1 AspNetUsers

The Identity system automatically creates this table, which stores user data, including usernames, email addresses, and other authentication information. The system stores passwords in a secure, hashed form. Although I am not currently using all the fields in the application, they may become useful additions in future developments.

4.3.2 AspNetUserRoles

This table stores the relationship between users and their roles in the Identity system. It has a single purpose: to map users to a role. It contains two fields: the user ID and the role ID.

4.3.3 AspNetRoles

The AspNetRoles table stores the roles used by the Identity system, assigning a unique ID and name to each role. Currently, there are three roles defined in: Administrator, Manager, and Member. These roles determine the access level and

permissions of users within the application, allowing **administrators to restrict** access to distinct functions based on role.

Id [PK] text	Name character varying (256)	NormalizedName character varying (256)	ConcurrencyStamp text
b3ed9eda-05a6-4411-96a1-2e018ae2301a	Admin	ADMIN	[null]
ea738c3f-d455-428b-85fc-cc6a718d9b80	Manager	MANAGER	[null]
5ab95710-45b1-4ae0-8d2a-283e64b472f9	Member	MEMBER	[null]

Figure 14: Contents of the AspNetRoles table

4.3.4 Grave

The Grave table keeps track of graves. Each grave has a type and three identifiers that indicate which section of the cemetery the grave is in, which row it is in, and which plot it is in within that row.

4.3.5 Deceased

This table stores data about the deceased, such as their names, dates of birth and death, and the ID of the grave containing their remains. This table is key to identifying and managing the deceased.

4.3.6 Message

I store memorial messages in this table. Each message has a type, author, message text, and creation date. This allows relatives and other mourners to share their memories in memory of the deceased.

4.3.7 Polygon

The Polygon table stores the geometric data of structures that the map displays. The StructureType field specifies the type of structure marked by the polygon. Currently, the system only displays graves on the map, but in the future, it will also be able to display memorials, funeral homes, and other buildings.

4.3.8 Point

I store specific coordinate points in the cemetery in this table. For each point, I assign the geographical latitude (Lat) and longitude (Lng) values, as well as which polygon they are part of. This ensures that the system displays the cemetery map accurately.

4.4 Client application architecture

When designing the application, my main goal was to clearly separate the individual functional parts and to make the project well-structured and transparent.

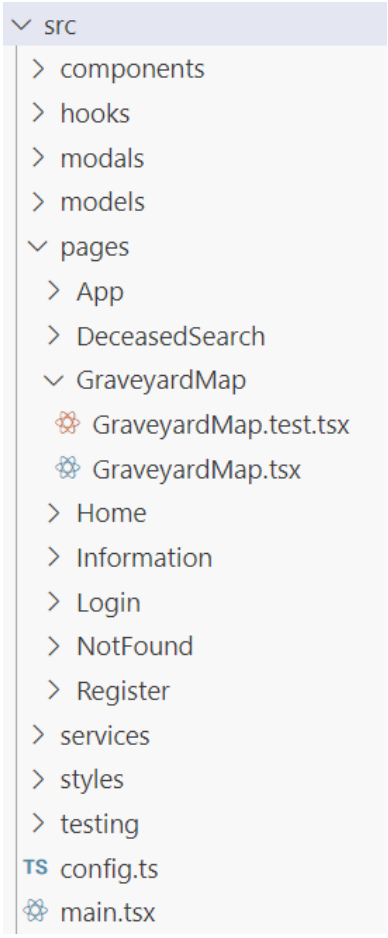

Figure 15: Structure of the client application

I collected the components of the application in the "components" folder. Within this folder, I created a system that groups them according to their functions. I paid particular attention to the reusability of each component, so I could use them in multiple places within the application. I also grouped these components in one place in the Shared folder.

The individual pages are in the "pages" folder. I gave pop-up windows (modals) a separate place in the application structure, considering that they perform most of the data management operations and there are a relatively large number of them in the application.

I also organised the operational logic of the programme, such as data models and diverse services, into separate modules. Data models ensure that the system presents the data used in the application in a uniform format and structure, while services help to separate individual operations, such as network communication, from display.

I collected the styles associated with different display elements, such as the appearance of forms or pop-up windows, in a central location. This allowed me to ensure that their appearance was consistent.

5 Implementation

In this chapter, I describe the operation of the application and the solutions used during development, using specific examples and highlighted use cases.

5.1 Styling

I used the Chakra UI component library to display the website. I used so-called style parameters (Style Props) to style the components, as this is a convenient way to specify style-related settings directly within the component. Below is an example of a button that toggles the text display of the password.

```
<Button
  onClick={() => setShowPassword(!showPassword)}
  variant="ghost"
  size="md"
  fontSize="x-large"
>
  {showPassword ? <ViewOffIcon /> : <ViewIcon />}
</Button>
```

Style parameters allow you to customise individual components, such as colours, margins, shadows, and font sizes, all within JSX code, making the code shorter and easier to read. This approach is particularly useful when developers need to change styles dynamically based on certain conditions, such as the state of a button (_hover - when the mouse pointer is over it).

However, this solution also has a disadvantage: styling is not separated from the JSX code, which is not always beneficial. For example, in the Angular framework, styling should be stored in separate files, which better follows the principle of separation of concerns.

5.2 User management

User management is essential in any registration system, as it allows for the separation of different permission levels, especially for users with administrator rights.

However, in the case of the application I developed, one of the main goals of the design process was to make as many features as possible available to unregistered users.

This approach was motivated by the fact that the primary target audience of the application is people living in rural areas, who in many cases do not have an email address that they could use for registration and whose digital literacy is limited. For such users, simplifying access and allowing them to use the application without logging in significantly increases its usability.

5.2.1 Login

Since registration is not required to use the application, the login page does not appear as the home page but is accessible from the address bar. For unauthenticated users, the "Login" menu item is visible in the address bar, otherwise the "Logout" menu item is visible.

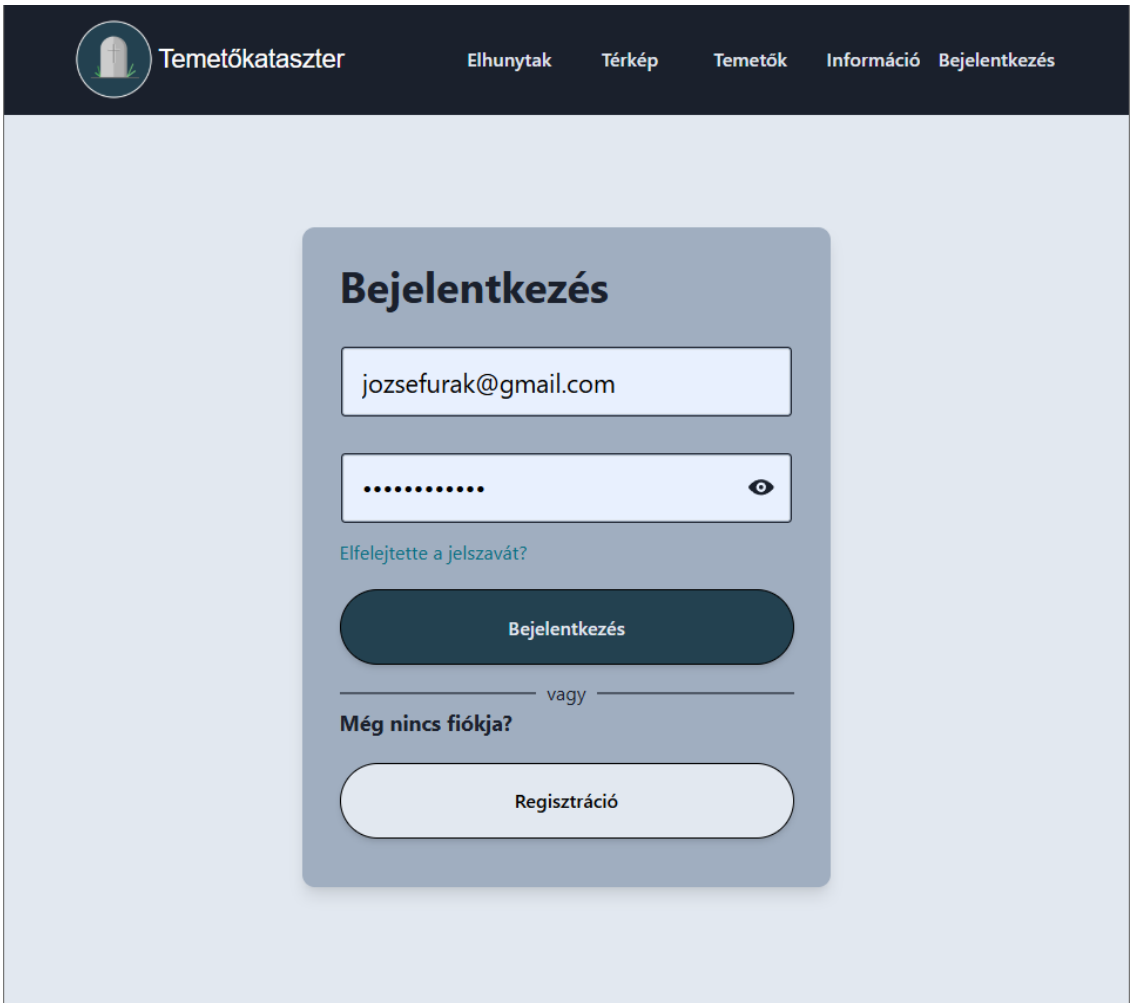

Figure 16: Login interface

The code snippet below creates the password input field. On the right side of the input field, there is a button with an icon that toggles between showing and hiding the

password. The field always displays the current value of the password useState type state variable.

```
<InputGroup mb="20px">
  <Input
    type={showPassword ? "text" : "password"}
    placeholder="Password"
    value={password}
    onChange={(e) => setPassword(e.target.value)}
    {...inputStyle}
  />
  <InputRightElement alignContent="centre" verticalAlign="centre"
    height="100%" mr="10px">
    <Button
      onClick={() => setShowPassword(!showPassword)}
      variant="ghost" size="md" fontSize="x-large"
    >
      {showPassword ? <ViewOffIcon /> : <ViewIcon />}
    </Button>
  </InputRightElement>
</InputGroup>
```

When logging in, I send a POST request to the API with Axios, asynchronously of course, which I track using useMutation. If the response is successful, I save the user data to sessionStorage and display a success toast notification. When an error occurs, I display a corresponding notification, considering the Axios response status code. Based on the status field of the isLoading mutation, I also display a loading animation (spinner) on the page.

```
const loginMutation = useMutation(loginUser, {
    onSuccess: (data: User) => {
      sessionStorage.setItem("username", data.username);
      sessionStorage.setItem("role", data.role);
      toast({
        title: "Login successful", description: "Welcome!",
        status: "success", duration: 5000, isClosable: true,
      });
      setUser(data);
      navigate("/");
    },
    onError: (error: AxiosError) => {
      const errorMessage =
        error.response && error.response.status === 401
          ? "Invalid username or password."
          : "Something went wrong. Please try again later.";
      toast({
```

```
        title:
          error.response && error.response.status === 401
            ? "Login failed": "An error occurred",
        description: errorMessage, status: "error",
        duration: 5000, isClosable: true,
      });
    },
  });
```

Figure 17: Failed login warning in case of incorrect data

I use ASP.NET Core Identity for user management. Although I initially tried to use the built-in Identity API available since .NET eight, I realised that I wanted to achieve a different functionality, so I created my own endpoints. The Controller is a "Thin Controller" that does little else but call the appropriate method of the service, as shown below.

```
public async Task<LoginResponseDTO> LoginAsync(LoginDTO model)
{
  IdentityUser? user = null;
  if (model.UserIdentifier.Contains("@"))
  {
    user = await _userManager.FindByEmailAsync(model.UserIdentifier);
  }
  else
  {
    user = await _userManager.FindByNameAsync(model.UserIdentifier);
  }
  if (user == null)
  {
    return new LoginResponseDTO {Succeeded=false, Message="User not found"};
  }

  var result = await _signInManager
    .PasswordSignInAsync(user, model.Password, false, false);
  if (result.Succeeded)
  {
    var roles = await _userManager.GetRolesAsync(user);
    string role = roles.FirstOrDefault() ?? "Member";
    var response = new LoginResponseDTO
    {
      Succeeded = true,
      Message = "Login successful",
      Username = user.UserName,
      Role = role
    };
    return response;
  }
  return new LoginResponseDTO{Succeeded=false,Message="Invalid Login"};
}
```

The UserIdentifier parameter can contain either a username or an email address, so the function checks this first. In the next step, it searches for the corresponding user based on this information. Once the user has been found, the system performs the login process and retrieves the user's role. The function then returns the user's data in a LoginResponseDTO type object. If any step fails (for example, the user cannot be found or the password is incorrect), the function returns an error message.

After successful login, a session cookie is stored in the browser to identify the user. This cookie is automatically sent with every subsequent API request until the session ends. This allows the user to remain identified for further requests. The session ends when the user logs out or closes their browser.

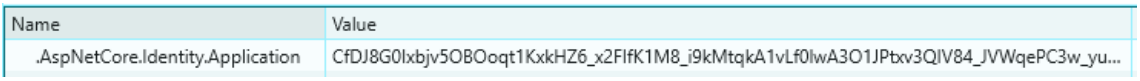

Figure 18: Session cookie stored in the browser

5.2.2 Registration

The registration process is remarkably like the login process, and the design of the two pages is identical. The only difference is that the registration page includes additional fields for confirming the username and password. It also uses the same API but calls a different endpoint.

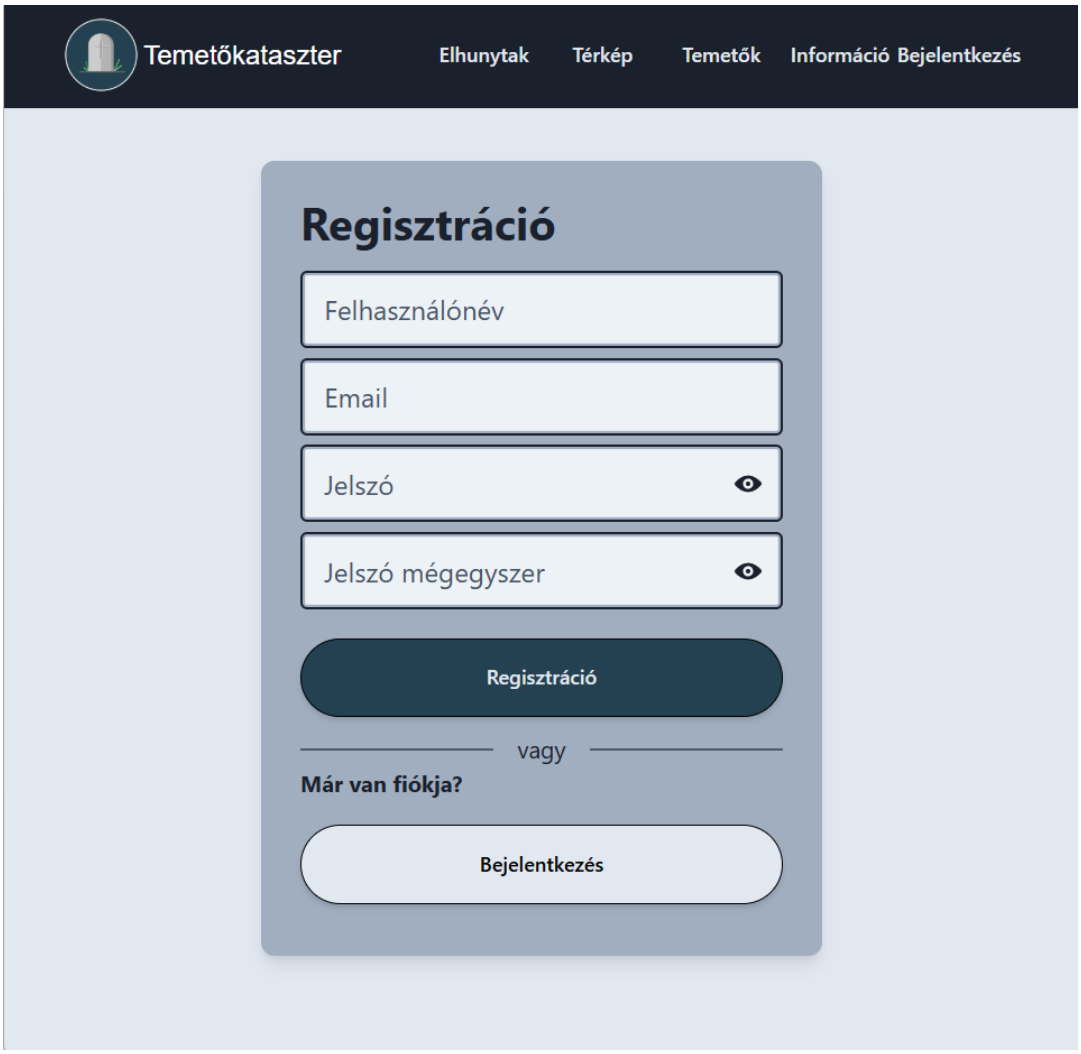

Figure 19: Registration interface

Upon successful registration, Identity automatically adds the user's data to the AspNetUsers table. It does not store the password as plain text, but first encrypts it using a one-way algorithm, ensuring that the original password cannot be recovered. Identity uses the PBKDF2 (Password-Based Key Derivation Function 2) algorithm to hash passwords, repeating the hashing process through 100,000 iterations. This method provides outstanding security and protects user data even if the database is compromised[18] .

Id [PK] text	UserName character varying (256)	Email character varying (256)	PasswordHash text
f6c2d7b5-f0a9-4ed6-9503-c47c530cbf16	test	test@test.com	AQAAAAIAAYagAAAAEGjG7ioE1u7xZG2+8LONO9jssORjPtvU...

Figure 20: Example of a user database record

5.2.3 Authorisation control

The code snippet below shows the application's authorisation management settings, which ensure that different user roles have the appropriate access. Authorisation policies determine which user groups can access certain resources or perform different operations in the application. For example, administrators have full access, while managers and members have limited permissions.

```
builder.Services.AddAuthorization(options =>
{
  options.AddPolicy("Admin", policy => policy.RequireRole("Admin"));
  options.AddPolicy("Manager", policy => policy.RequireRole("Admin",
    "Manager"));
  options.AddPolicy("Member", policy => policy.RequireRole("Member",
    "Admin", "Manager"));
});
```

I protected each endpoint and API with the [Authorise] attribute, which ensures that only users with the appropriate permissions can access them. In the example below, I protect the given endpoint with the Manager policy, so only users with the "Manager" or "Admin" role can access it.

```
[Authorise(Policy = "Manager")]
```

Client-side authorisation is also crucial. This ensures that unauthorised users cannot view content or use features to which they are not entitled.

To this end, I created a custom user hook that allows us to access the logged-in user's data anywhere in the application. Using this hook, I created wrapper components that only display the content they surround if the user's role matches the required permission.

```
const AdminOnlyWrapper: React.FC<AdminOnlyWrapperProps> = ({ children })
=> {
  const { user } = useUser();

  if (user.role === "Admin") {
    return <React.Fragment>{children}</React.Fragment>;
  }
  return null;
};
```

5.3 Search page

The search page can be accessed by selecting the "Search" option in the header. Here, users can search for, filter and sort the deceased. The results are displayed in card

form and contain some information about the deceased, as well as a picture of the grave where they are laid to rest. If no picture is available, a placeholder image will appear in its place.

```
<Image
  src={IMAGES_URL + deceased.imageUrl}
  w="100%"
  fallbackSrc="https://placehold.co/720x540?text=Nincs+uploaded+image"
/>
```

The search functions include searching by name and filtering by year of death and birth. By entering the year of death, we can filter for people who died before the specified year. In addition, you can also search by year of birth, allowing you to find people who died in or after a given year. This allows you to search for people who lived in any period.

The search can also be based on terms appearing in the names, making it easy to find deceased persons whose names contain the searched term. By clicking on the search button, the deceased persons matching the search criteria will appear in the list below. The results can be sorted by name, date of birth or date of death. Furthermore, by clicking on the card, you can access the memorial message board for the deceased person.

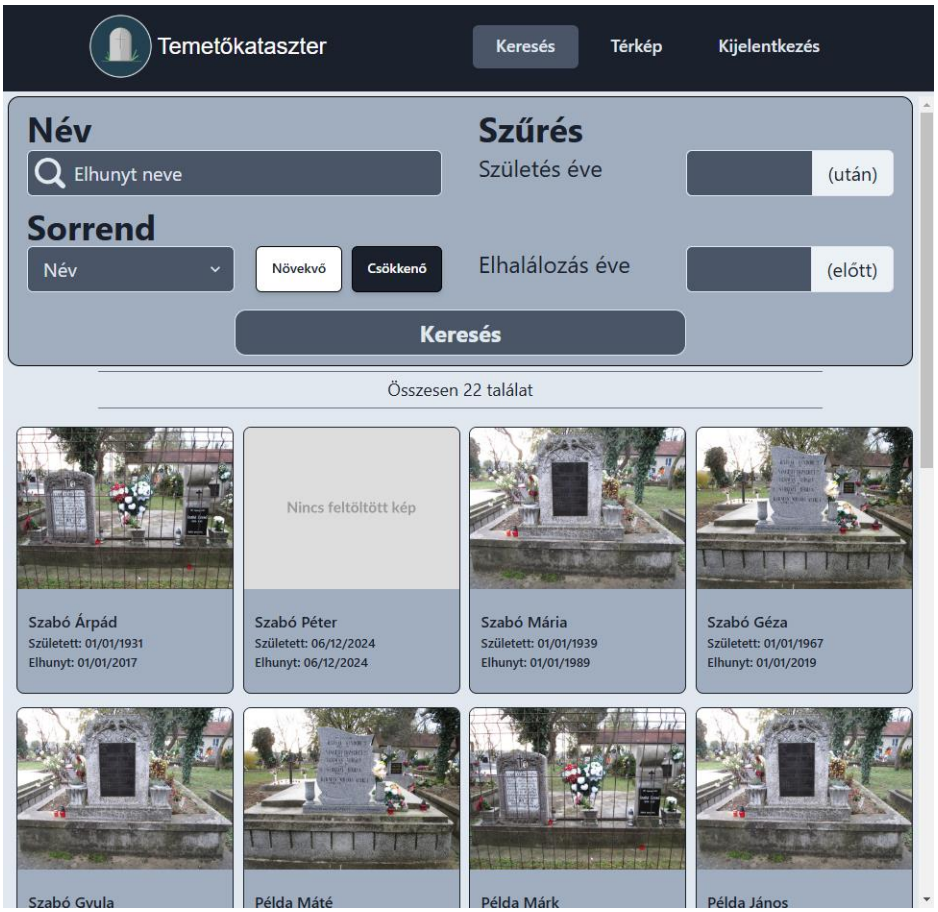

Figure 21: Sorting search results by name in descending order

I also used pagination on this page to avoid unnecessary network traffic. I pass the pagination parameters, such as the current page, page size, and total number of results, in the request header.

```
int pageNumber = int.TryParse(Request.Headers["Page-Number"], out var
  parsedPageNumber) ? parsedPageNumber : 1;
int pageSize = int.TryParse(Request.Headers["Page-Size"], out var
  parsedPageSize) ? parsedPageSize : 20;

var (deceasedItems, totalCount) = _deceasedService.SearchDeceaseds(name,
  birthYearAfter, deceaseYearBefore, orderBy, pageNumber, pageSize);

Response.Headers.Append("Total-Count", totalCount.ToString());
Response.Headers.Append("Page-Number", pageNumber.ToString());
Response.Headers.Append("Page-Size", pageSize.ToString());
Response.Headers.Append("Total-Pages",
  ((int)Math.Ceiling((double)totalCount / pageSize)).ToString());

Response.Headers.Append("Access-Control-Expose-Headers", "Total-Count, Page-
Number, Page-Size, Total-Pages");
```

5.4 Memorial page

The purpose of the memorial page is to provide an opportunity to remember the deceased, allowing relatives, friends, and acquaintances to pay their respects to the memory of the deceased. In the centre of the page is a message board where visitors can share their messages. Each message can also be accompanied by a commemorative item.

On the left side of the page is the same component used on the search page, consisting of the search box and the list of results. This allows you to navigate to other memorial pages, giving you quick access to the profiles of other deceased persons. The right side of the page displays detailed information about the deceased person and a picture of their grave.

The page also allows you to post your own memorial message on the wall. In addition, administrators have access to an editing option that allows them to modify the details of the deceased. The addition and editing process also takes place in a modal window, which simplifies the execution of operations, ensuring a user-friendly experience and smooth data management.

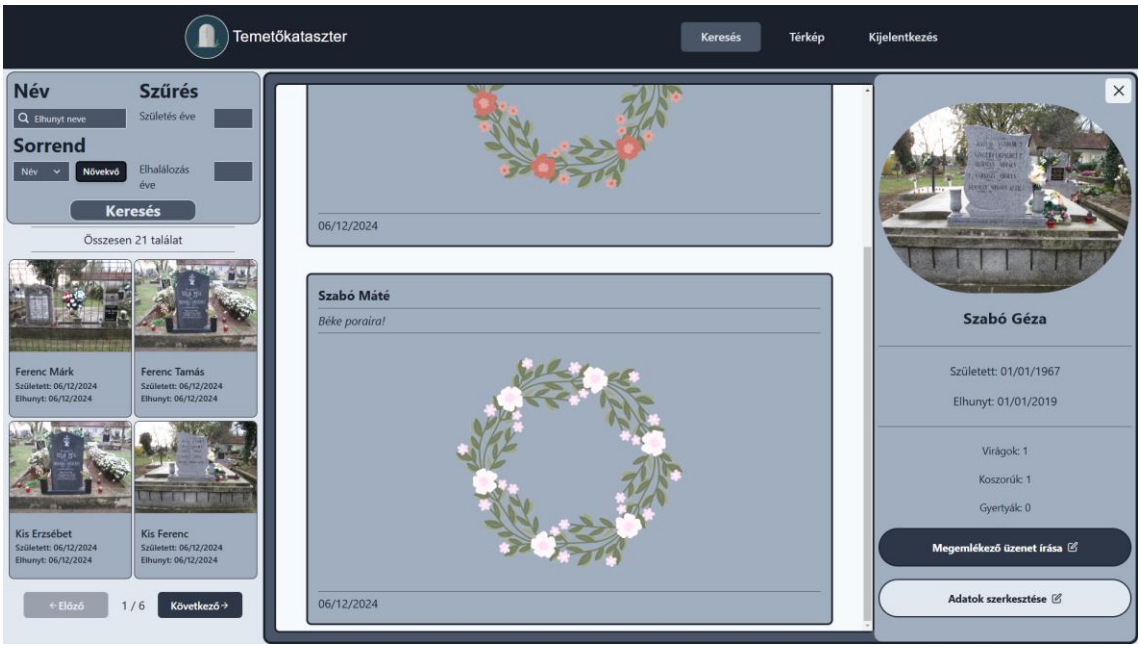

Figure 22: Memorial page

5.5 Map display

5.5.1 React Leaflet

I chose the React Leaflet library to display the cemetery map because it has several advantages that make it ideal for my project. React Leaflet is a React-specific wrapper for the Leaflet JavaScript library that makes developing map applications easier and more efficient. There are several key factors behind my choice.

The first and most important consideration was React integration. React Leaflet allows for seamless integration of Leaflet maps and React applications because it uses a component-based development model. Different map layers, pins, polygons, and other interactive elements can be treated as React components. For example, the code snippet below creates a polygon on the map.

```
<Polygon key={layer.id} positions={layer.latLngs} />
```

Furthermore, React Leaflet offers a wide range of plugins. For example, the React-Leaflet-Draw plugin I used allows users to draw new elements on the map, which was an essential feature for the cemetery map as it allows graves to be marked.

Extensive community support and documentation: As React Leaflet is the most well-known React-based wrapper for the Leaflet library, there is extensive community

support, forums, and tutorials available. The React developer community is active, so solutions to any problems that arise can be found quickly.

Overall, React Leaflet was the ideal choice for displaying the cemetery map, as it provides the necessary functionality, flexibility, and easy integration into the React application. The library enabled rapid development and easy implementation of the necessary interactive map elements.

5.5.2 Custom map

Leaflet is designed for use with world maps, such as OpenStreetMap. However, in my application, I wanted to use a custom map, specifically a drone image of the cemetery. To do this, I had to find a solution that would allow me to integrate the drone image as a map into the Leaflet environment.

This was necessary because the publicly available satellite images did not provide sufficient detail. Fortunately, I happened to have my own drone footage of the cemetery taken specifically for this purpose. This allowed me to provide a much more detailed map, which significantly improves the user experience.

Figure 23: Detail from the drone image

The entire image is 4.5 gigabytes in size, which is too large to load directly into a browser. Furthermore, a standard screen cannot display such a high-detail image for the entire map. The detail only becomes incredibly useful with significant magnification.

Therefore, I use tiling to display the map. This allows me to provide clients with images that have the appropriate resolution for the given zoom level, thus avoiding unnecessary traffic.

5.5.3 Tile generation

The drone footage was captured in ECW (Enhanced Compression Wavelet) format, which is a special compressed image file format primarily used in geospatial applications to store high-resolution images. I first had to convert it to PNG format. Unfortunately, the conversion process resulted in a significant loss of quality, but I will look for a better solution in the future to preserve the quality of the image.

I used Adobe Photoshop to process the drone footage and create the tiles from the resulting image. In Photoshop, I ran a publicly available GitHub script for slicing tiles[19] .

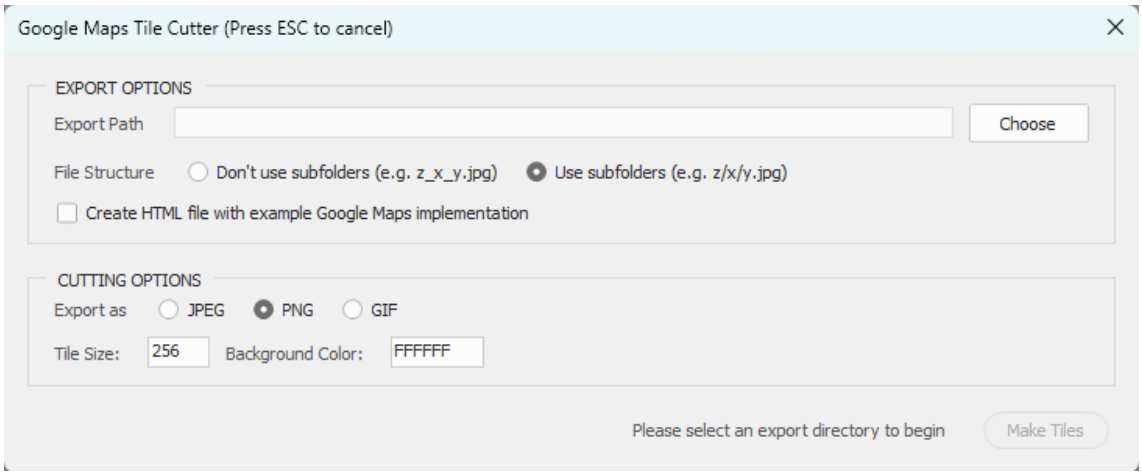

Figure 24: Tile slicing script settings

The image on which the script is run is exported in the folder structure shown below. Within this, the tiles at different zoom levels will be placed in additional folders from the original map.

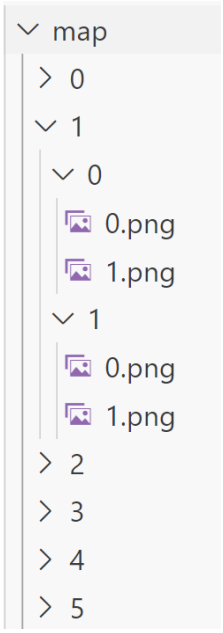

Figure 25: Folder structure containing tiles

The higher the resolution of the image, the more tile levels can be produced from it. From the image I used (21200x22800 pixels), I produced six levels. Each level has four times as many tiles as the previous one. The higher the zoom level, the more detailed and larger the images are.

5.5.4 Map display

To display the map, I used React Leaflet components and configured them to achieve the desired functionality. I used the MapContainer component as the map container, which is essential for interactivity and proper map display.

```
<MapContainer
  center={centre}
  zoom={3}
  scrollWheelZoom={true}
  placeholder={<MapPlaceholder />}
  maxZoom={10}
  minZoom={3}
  boundsOptions={{ padding: [0, 0] }}
  maxBounds={myBounds}
  maxBoundsViscosity={100}
>
  { /* Content */ }
</MapContainer>
```

I set the minimum allowed zoom level (minZoom) to three to ensure that the map always fills the container and cannot be zoomed in to such a small level that the map is

surrounded by empty space. I limited the maximum zoom level to ten because at higher levels, the map resolution is no longer of sufficient quality and becomes distorted. Furthermore, I cannot imagine a scenario where a higher zoom level would be necessary.

To display the drone footage, I used the TileLayer component, which allows map tiles to be loaded into the container. I created six zoom level tiles from the image I used, which I also had to specify in the TileLayer configuration.

Since suitable tiles are only available at zoom levels lower than seven, if you zoom in further on the map, the application will still use the largest tiles, i.e. those at level six. Of course, there will be a noticeable deterioration in quality at zoom levels seven and above.

I also prevented the map from rotating, meaning that the user cannot go beyond the edges of the map, where they would find the same map again. This would be a useful feature for a world map, but not in this case. I left the attribution field blank because I am not using a public map, but tiles generated from my own drone footage.

The tiles are served by the client application from the public folder. The image associated with a given tile is loaded from the URL "/map/{z}/{x}/{y}.png", where "z" denotes the zoom level, "x" denotes the horizontal position of the tile within that level, and "y" denotes the vertical position of the tile. This follows the folder structure containing the tiles, so it will successfully find the image if it exists. This way, the map displays the image fragment corresponding to the given zoom level and coordinates, enabling efficient and detailed map display.

```
<TileLayer
  minNativeZoom={0}
  maxNativeZoom={6}
  noWrap
  attribution=""
  url="/map/{z}/{x}/{y}.png"
/>
```

I also placed the EditControl component from the react-leaflet-draw library inside the map container. Since I only needed to draw polygons, I disabled all other geometric drawing functions. In addition, I did not need to edit polygons, and I developed my own solution for deleting them, so I disabled these functions of EditControl as well. Only users with manager privileges can designate new graves, so I surrounded this component with the appropriate wrapper component.

```
<ManagerOnlyWrapper>
  <EditControl
    position="topright"
    draw={{
      polygon: true, rectangle: false, polyline: false,
      circle: false, marker: false, circlemarker: false,
    }}
    edit={{ edit: false, remove: false }}
    onCreated={_onCreate}
  />
</ManagerOnlyWrapper>
```

5.5.5 Displaying graves on the map

When navigating to the map page, I request the coordinates of the polygons from the backend. I need to convert the data I receive into a format that Leaflet can display as polygons. If an error occurs during loading, I alert the user with a toast notification. While waiting for the data to arrive, I display a loading overlay on top of the map based on the isLoading status to indicate that loading is in progress.

```
const { isLoading } = useQuery("polygon", retrievePolygons, {
  onSuccess: (polygons) => {
    const formattedPolygons = transformPolygons(polygons);
    setPolygons(formattedPolygons);
  },
  onError: () => {
    toast({
      title: "An error occurred while loading polygons",
      status: "error", duration: 3000, isClosable: true,
    });
  },
});
```

I use React Leaflet's Polygon component to display the polygons marking the graves. I determine the correct placement of the polygons based on data received from the backend. I also assign a reference to the Polygon component, which allows me to dynamically modify the properties of the polygon later. I use this reference object to modify the style of the polygon, for example when the user moves the cursor over or away from the polygon.

```
<Polygon
  key={layer.id}
  positions={layer.latLngs}
  ref={polygonRef}
  eventHandlers={{
```

```
    mouseover: highlightFeature,
    mouseout: unHighlightFeature,
    click: handleClick,
  }}
  pathOptions={{
    colour: isSelected ? "red" : getPolygonColour(layer.structureType),
    fillOpacity: 0.3, opacity: 1, weight: 2,
  }}
>
  <GravePopup
    grave={grave}
    graveType={layer.structureType}
  />
</Polygon>
```

The colour of the polygon displayed varies depending on the type of grave. The seven diverse types of graves include covered graves, framed graves, burial mounds, memorial pillars, planned graves, crypts, and urn walls. The three most common types, namely covered graves, framed graves and burial mounds, are coloured cyan, green, and orange, respectively.

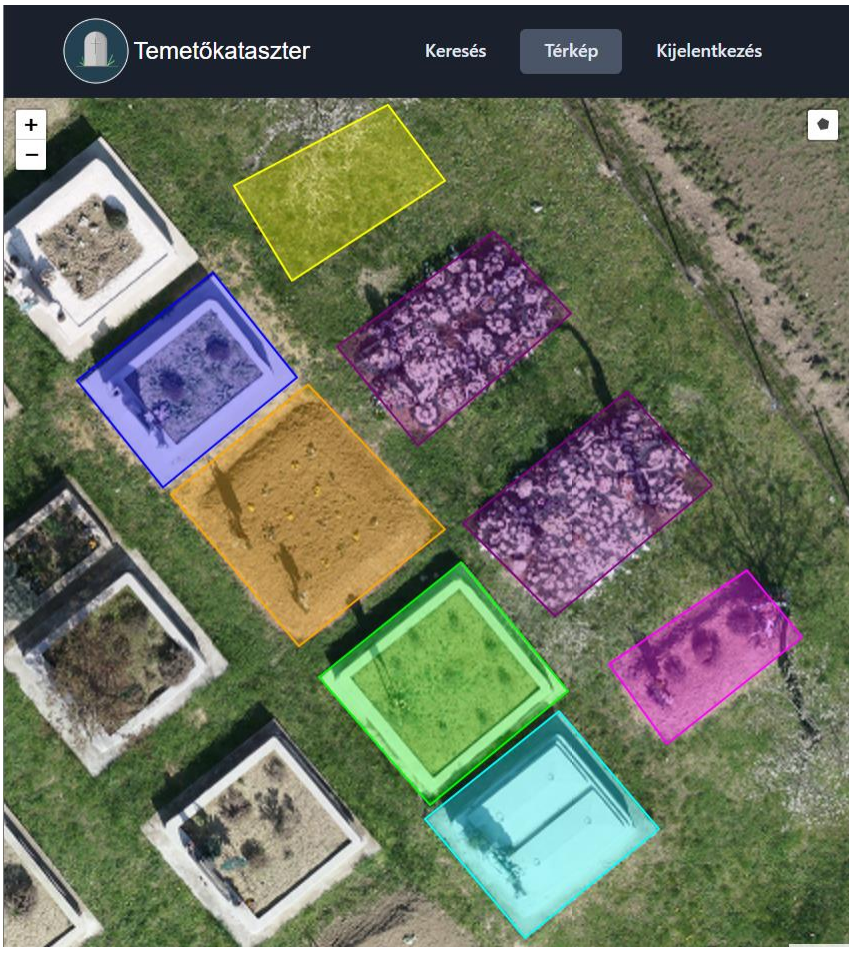

Figure 26: Appearance of all grave types

When the user clicks on a polygon marking a grave, the system retrieves the detailed data of the corresponding grave from the backend. A pop-up window then appears, listing the information about the deceased buried in that grave, the type of grave, and a picture of the grave, the grave site identifier, is also displayed, which uniquely identifies which cemetery plot the grave site belongs to, which row it is in, and the plot number.

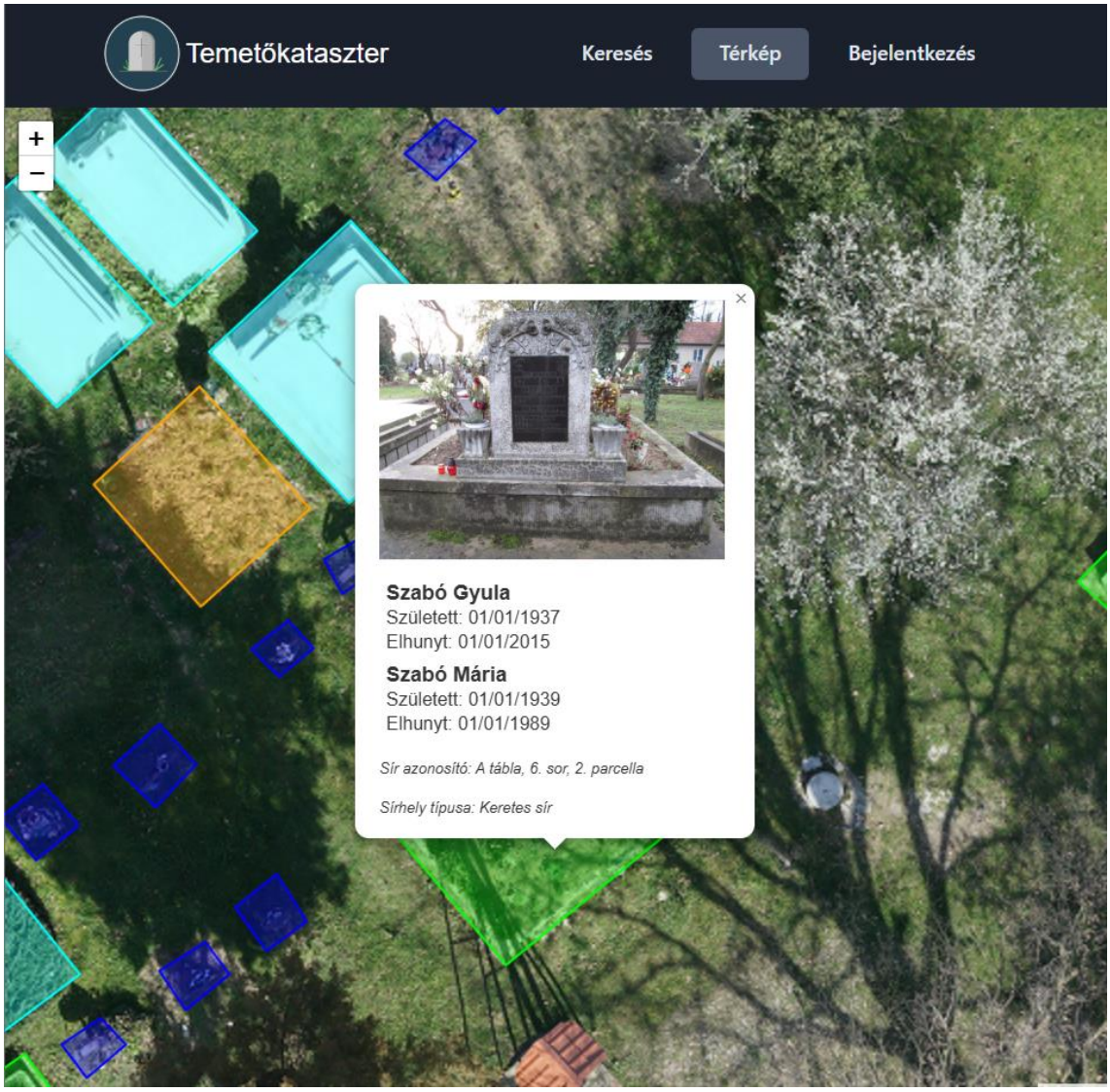

Figure 27: Pop-up window displaying information about the deceased

5.5.6 Cemetery caretaker functions

For cemetery caretakers, i.e. users with manager privileges, the pop-up window allows them not only to view information, but also to modify it. The grave deletion function is available here, which, through cascading deletion, deletes not only the grave, but also the stored polygon marking it and all deceased persons associated with that grave

from the database. In addition, cemetery caretakers can use this window to edit the list of deceased persons buried in the grave: they can delete names from the list or add new deceased persons.

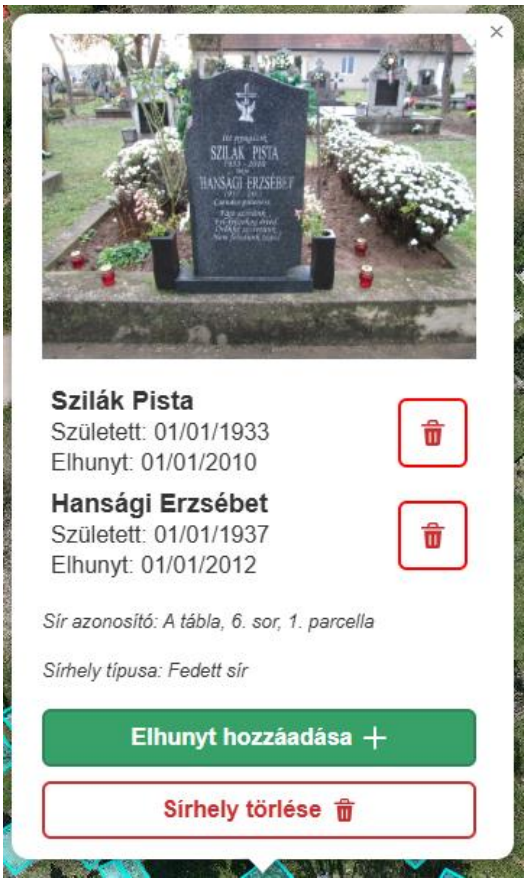

Figure 28: Contents of the pop-up window for users with manager privileges

Before the operation is performed, the deletion of the grave or the deceased must be confirmed in a confirmation window to avoid accidental removal. Only when the deletion is confirmed is the deletion request sent to the backend.

The component containing the modal window must be given a callback function that it can use to notify the parent component. This callback function updates the parent component's state so that it can remove the deleted deceased person from the displayed list.

```
<DeceasedDeleteModal
  isOpen={isDeceasedDeleteOpen}
  onClose={() => setIsDeceasedDeleteOpen(false)}
  deceased={selectedDeceased}
  notifyDeceasedDeleted={() =>
    handleDeceasedDeleted(selectedDeceased.id)
  } />
```

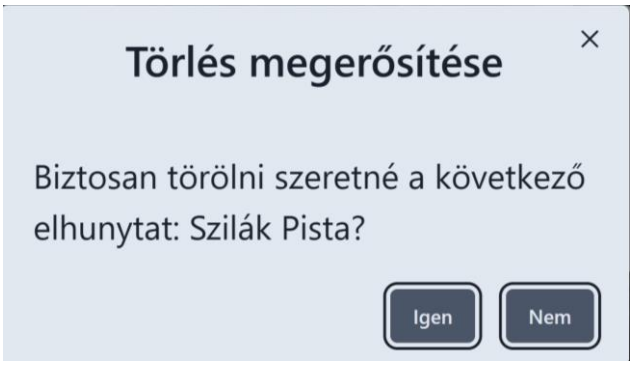

Figure 29: Confirmation of deletion modal window

I used the React Leaflet Draw library to mark new graves on the map. To start drawing, click on the polygon icon in the upper right corner of the map container. This button is only available to users with manager privileges.

Once drawing mode is activated, you can mark the corners of the grave on the map. Although it is possible to mark more than four nodes, this is not recommended for consistency and visual clarity. However, in the case of a burial mound or a more complex crypt, it is unavoidable. To finish drawing, the user can click on the "Finish" button or click on the first marked point again to close the polygon.

Figure 30: Drawing a polygon to mark a grave

Once the polygon has been drawn, a modal window for creating the grave appears. In this window, the type of grave, the letter of the corresponding table, its row, and the

plot must be specified. These three pieces of information must be unique, which is ensured by a database index.

```
[Index(nameof(Table), nameof(Row), nameof(Parcel), IsUnique = true)]
```

In addition, there is also the option to upload an image of the grave. I send the image in FormData format to the backend application, where the system stores it in the wwwroot folder. After uploading the image, the access path pointing to the URL of the uploaded image is entered into the ImageUrl field of the grave. My goal was to eliminate the unnecessary data traffic that would occur if the image itself had to be continuously transmitted.

```
if (dto.Image != null)
{
  var imagePath = Path.Combine("wwwroot", "images", dto.Image.FileName);
  using (var stream = new FileStream(imagePath, FileMode.Create))
  {
    await dto.Image.CopyToAsync(stream);
  }
  grave.ImageUrl = $"/{dto.Image.FileName}";
}
```

The search box on the left side of the map is almost identical to the search page and uses the same component, so it works the same way. The list below the search box shows the deceased who meet the criteria, and their graves are highlighted in red on the map, making them easy to find.

Figure 31: Successful search results

6 Evaluation

6.1 Experience

While authoring my thesis, I learned about several new technological solutions that I did not have the opportunity to fully understand during my university studies. I realised that it is the larger, more complex projects that really deepen our knowledge and give us the opportunity to apply theory at a practical level.

In the field of web development, it is noticeable that client-side systems are developing at a rapid pace and modern technologies are constantly emerging. However, during my thesis, I learned basic patterns and principles that I believe will be fully applicable not only to today's technologies but also to future systems.

The biggest challenge for me in authoring my thesis was how to build a large-scale project in a maintainable and well-structured way. In such a complex system, it is particularly important to break the project down into smaller, clearly separable parts that perform clearly defined tasks and can be easily modified and expanded in the future.

I realised that a modular structure and clear boundaries between the individual components are not only a prerequisite for transparency, but also enable more efficient development, debugging and even independent development of distinct parts of the project. In applying these principles, I learned a lot about how proper design can increase the long-term sustainability of a system.

My most important realisation was that web development is an extremely attractive field for me, and it is closest to my abilities and interests.

6.2 Opportunities for further development

While preparing this thesis, I noticed several areas for improvement that could further increase the efficiency, security, and usability of the system. The most key area for improvement would be to ensure the responsiveness of the website. Nowadays, it is essential that a web application can be used on most devices: mobile phones, tablets, different screen sizes in the same way as on a desktop computer.

To determine the exact location of graves, it would be essential to be able to import the coordinates of the corners of the graves directly from an instrumental survey

conducted in the field. This would allow for the accurate marking of graves covered by vegetation, as these are not visible on drone or satellite images.

Another key step is to improve the security of user interactions, with particular attention to filtering out obscene, offensive, or malicious messages. The current implementation does not include any automatic filters for this type of content.

In this regard, preventing over-posting attacks is also of paramount importance. Such attacks usually lead to system overload, so it would be worthwhile to introduce mechanisms that can limit the number of messages users can send within a given period or warn them if they show excessive activity.

To make the system more user-friendly, it would also be advisable to introduce a password reminder email service. This feature will allow users to easily restore access to their accounts if they forget their passwords.

7 References

[1] Microsoft, „What is .NET?," [Online]. Available: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet. [Hozzáférés dátuma: 1 November 2024].

[2] Microsoft, „A .NET bemutatása," [Online]. Available: https://learn.microsoft.com/hu-hu/dotnet/core/introduction. [Hozzáférés dátuma: 2 November 2024].

[3] J. P. Smith, „ASP.NET Core Web API architecture," [Online]. Available: https://www.thereformedprogrammer.net/how-to-write-good-testable-asp-net-core-web-api-code-quickly/. [Hozzáférés dátuma: 8 November 2024].

[4] Microsoft, [Online]. Available: https://learn.microsoft.com/en-us/ef/core/. [Hozzáférés dátuma: 9 November 2024].

[5] S. Getachew, „Entity Framework Core Latest Version for Beginners: Mapping Your First Database," [Online]. Available: https://medium.com/@solomongetachew112/entity-framework-core-latest-version-for-beginners-mapping-your-first-database-ef7964462fe6. [Hozzáférés dátuma: 9 November 2024].

[6] Microsoft, „Introduction to Identity on ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-9.0&tabs=visual-studio. [Hozzáférés dátuma: 9 November 2024].

[7] TypeScript, [Online]. Available: https://www.typescriptlang.org/. [Hozzáférés dátuma: 7 November 2024].

[8] A. Camus, „Introduction to ReactJS: A Guide for Beginners," [Online]. Available: https://www.microverse.org/blog/introduction-to-reactjs-a-guide-for-beginners. [Hozzáférés dátuma: 11 November 2024].

[9] I. M. Dadzie, „Document Object Module in JavaScript,” [Online]. Available: https://www.linkedin.com/pulse/document-object-module-javascript-isu-mwurf/. [Hozzáférés dátuma: 11 November 2024].

[10] Chakra, „Chakra Dokumentáció,” [Online]. Available: https://v2.chakra-ui.com/getting-started. [Hozzáférés dátuma: 20 November 2024].

[11] R. C. Martin, „The Clean Architecture,” [Online]. Available: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. [Hozzáférés dátuma: 20 November 2024].

[12] HiBit, „Domain Driven Design: Layers,” [Online]. Available: https://www.hibit.dev/posts/15/domain-driven-design-layers. [Hozzáférés dátuma: 25 November 2024].

[13] Microsoft, „Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10),” [Online]. Available: https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application. [Hozzáférés dátuma: 1 12 2024].

[14] M. S. Arfa, „Implementing the Repository Pattern in C# and .NET,” [Online]. Available: https://www.linkedin.com/pulse/implementing-repository-pattern-c-net-mahdi-shayesteh-arfa/. [Hozzáférés dátuma: 1 December 2024].

[15] A. Karabulut, „Understanding Clean Architecture and Domain-Driven Design (DDD),” [Online]. Available: https://medium.com/bimar-teknoloji/understanding-clean-architecture-and-domain-driven-design-ddd-24e89caabc40. [Hozzáférés dátuma: 4 12 2024].

[16] Okta, „Data Transfer Object DTO Definition and Usage,” [Online]. Available: https://www.okta.com/identity-101/dto/#:~:text=A%20data%20transfer%20object%20(DTO,that%20carries%20data%20between%20processes.. [Hozzáférés dátuma: 3 December 2024].

[17] A. Beak, „Thin controllers and thin models,” [Online]. Available: https://www.linkedin.com/pulse/thin-controllers-models-andy-beak/. [Hozzáférés dátuma: 25 11 2024].

[18] Microsoft, „Hash passwords in ASP.NET Core,” [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/data-protection/consumer-apis/password-hashing?view=aspnetcore-8.0. [Hozzáférés dátuma: 4 12 2024].

[19] Bramus, „Photoshop Google Maps Tile Cutter Script (Google Maps Tile Generator),” [Online]. Available: https://www.bram.us/2012/04/23/photoshop-google-maps-tile-cutter-script/. [Hozzáférés dátuma: 5 12 2024].