BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE - ENGLISH

DIPLOMA THESIS

Attractions

Supervisor
Lect. dr. Pop Andreea Diana

Author
Pătrulescu Ronald Sandrino

2024

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ - ENGLEZĂ

LUCRARE DE LICENŢĂ

Atracții

Conducător științific
Lect. dr. Pop Andreea Diana

Absolvent
Pătrulescu Ronald Sandrino

2024

ABSTRACT

With the increasing popularity of social media and the growing interest in travel, there is a need for innovative features in travel-related applications. This bachelor thesis presents the design and implementation of a web application that provides a platform that allows visualisation of attractions and the ability to create your own attractions or lists of such attractions.

The first chapter contains the motivation, some historical background for web applications and the objective of this application.

The second chapter explores the theoretical notions relevant to web application development.

The third chapter presents the technical side of things and explains the frameworks and the technology stack used in developing the application.

The fourth chapter is a window that provides a view on the implementation details.

The fifth and final chapter provide a conclusion and some ideas about how the application could be improved.

Contents

Chapter 1

Introduction

This chapter serves as an opening to the thesis, providing an overview of the motivation, background and objective behind the development of a travel-focused media web application. While it builds on the abstract and preface by delving deeper into the subject, it also serves as a prelude to the technical and detailed discussions that will follow in subsequent chapters. The chapter is structured into several sections, each focusing on a particular aspect of the project.

1.1 Motivation

Travel, a cornerstone of shared experiences in the journey of life, serves as a dynamic conduit for the creation of enduring connections, precious memories, and deep personal insights. This potent medium invites us closer to each other, nature, and our inner selves, fostering a broader understanding of the world and our place within it. The exploration of diverse cultures and environments cultivates empathy and presents new perspectives, fulfilling the higher tiers of Maslow's hierarchy of needs such as self-actualization and transcendence. Living in a world driven fundamentally by relationships and the joy of experiencing the vitality of various cultures, travel becomes the nucleus of our most meaningful experiences. This basic human instinct to share individual journeys extends from the primal storytelling around a fire to the contemporary global narratives propelled by technology, underlining our collective yearning for shared experiences. Despite superficial differences, travel reveals profound commonalities between cultures, such as the instinct to survive, the pursuit of pleasure over pain, and our fundamental needs, thereby fostering empathy and reinforcing our interconnectedness.

1.2 Background

Numerous ground-breaking developments in computer science and technology have influenced the progress of web application development. In order to comprehend how the proposed travel-focused web application was developed, it is important to understand the key turning points that led to the current state of web apps. This section covers those turning points in detail.

1.2.1 The Internet

The invention of the internet fundamentally changed how people interact, obtain information, and conduct business in the modern day. The late 1960s and early 1970s saw the collaboration of numerous scholars and institutions to create the internet, often known as the "network of networks." The US Department of Defense's Advanced Research Projects Agency (ARPA) first created it as a decentralized communication network called ARPANET. Establishing a strong and resilient network that could endure partial failures and continue operating in the event of a calamity or military attack was the main driving force behind its design. The internet grew over time, connecting colleges, research centers, and eventually people all over the world. [CK74]

1.2.2 The World Wide Web

The World Wide Web (WWW), also referred to as the web, has transformed how we access and interact with information and has become a crucial part of the internet. While employed by the European Organization for Nuclear Research (CERN) in the late 1980s, Sir Tim Berners-Lee created the World Wide Web. His goal was to develop a system that would let users browse and distribute materials via hypertext links. WorldWideWeb, the first web browser, and the first web server were released in 1990 by Berners-Lee. These developments created the framework for the internet as we know it today. Fundamental web technologies like HTML (Hypertext Markup Language), which is used to structure web pages, and HTTP (Hypertext Transfer Protocol), which is used to communicate between servers and clients, allowed the web to grow quickly and be widely used. It changed the internet into a networked platform for dynamic content, multimedia, e-commerce, social networking, and many other applications from a collection of static documents. [BLF99]

1.2.3 HTML

As the common markup language for developing web pages and web applications, HTML (Hypertext Markup Language) is a vital component of the World Wide Web. It gives content, including as text, photos, multimedia components, and hyperlinks, a structure for organization and layout. The structure and display of web publications are defined by tags in HTML, which gives developers the ability to specify headings, paragraphs, lists, tables, forms, and other features. HTML's extensive popularity and function as the foundation of the web are due in part to its ease of use and adaptability. [Mozc]

1.2.4 CSS and JavaScript

The layout and formatting of HTML documents are described using the style-sheet language known as CSS (Cascading Style Sheets). It gives site designers control over elements like layout, colors, fonts, and animations that affect how online pages look. The structure of the web page is separated from the design elements using CSS, which makes it simpler to change and maintain the styling over several pages. With CSS, web designers can make websites that are visually appealing and responsive to various screen sizes and devices. [Mozb]

On the other hand, JavaScript is a flexible programming language that allows for dynamic behavior and interactivity on web sites. It enables programmers to handle events, create functionality, modify HTML components, and interact with servers. For producing interactive elements like form validation, sliders, carousels, and interactive maps, JavaScript is frequently utilized in web development. By providing real-time updates, client-side data processing, and seamless interaction with web apps, it improves the user experience. [Mozd]

1.2.5 AJAX

Asynchronous JavaScript and XML (AJAX) is a set of web development tools that permits data to be exchanged asynchronously between a web browser and a server without requiring a page reload. To construct dynamic and interactive online applications, it mixes JavaScript, XML (or other data formats like JSON), and asynchronous communication. Because AJAX enables seamless data retrieval and display without interfering with the user's browsing experience, web developers may utilize it to construct responsive and interactive user interfaces. Jesse James Garrett is credited with popularizing AJAX in his seminal piece "Ajax: A New Approach to Web Applications," which appeared on the website of Adaptive Path in 2005. The article explored how AJAX, which enables more dynamic and responsive web

applications, has the potential to transform web development. [Gar]

1.2.6 Emergence of Web Development Frameworks and Libraries

Web development frameworks and libraries have emerged in recent years and have advanced quickly, altering how web applications are created. These frameworks and libraries give programmers a selection of instruments, pre-built frameworks, and reusable parts that speed up the programming process and increase productivity [SG14]. They make it possible for developers to concentrate more on application logic than on low-level technical details, which makes it easier to create web apps that are reliable, scalable, and packed with features.

The introduction of these frameworks and libraries has greatly streamlined web development by enabling programmers to use pre-existing code, adhere to best practices, and follow established patterns. Routing, data binding, component-based architecture, form validation, and API connectivity are just a few of the many functionalities they provide [Ste21]. These technologies help developers create web apps more quickly and easily because they abstract complicated activities and offer well-documented APIs.

The ability of web development frameworks and libraries to address issues like code organization, code reusability, performance optimization, and cross-browser compatibility is a major factor in their success. Additionally, they encourage cooperation and knowledge exchange among developers through vibrant online communities, in-depth documentation, and a robust ecosystem of plugins and extensions.

Over time, a number of significant frameworks and libraries—including Angular, React, Vue.js, Django, Ruby on Rails, and Laravel—emerged, each with unique advantages and traits. These frameworks have a big impact on the web development scene and have been widely embraced by developers. [Vue] [Dja] [Rub] [Lar]

1.2.7 Evolution of Database Technologies

Data management systems have become increasingly complex and effective as a result of tremendous evolution and innovation in the field of database technologies. This section examines the significant turning points in database technology development, highlighting noteworthy developments and their influence on the industry.

1. Hierarchical and Network Models

   Network and hierarchical modeling were popular in the early days of database systems. Data was represented using these models, which were respectively pioneered by IBM's Information Management System (IMS) and the CODASYL Data Model, in the form of trees or graphs [Dat03]. These models orga-

nized data well for the purposes of the applications they were designed for, but they lacked the adaptability to manage complicated relationships.

2. Relational Databases

   Edgar Codd changed the database industry in the 1970s by introducing the relational model [Cod70]. In relational databases, data was arranged into tables with rows and columns and relationships defined by keys. Relational database management systems (RDBMS) based on SQL, such as Oracle, MySQL, and PostgreSQL, were made possible by this model's ability to store and query structured data in a powerful and flexible manner.

3. Object-Oriented Databases

   As object-oriented programming languages became more popular, object-oriented databases (OODB) started to take the place of relational databases. By enabling the direct representation of objects in databases, OODBs sought to close the information storage and programming language gap [US90]. Complex data structures could be stored more easily as a result, and database models and application code were better integrated.

4. NoSQL Databases

   NoSQL (Not Only SQL) databases were created as a result of the development of web applications and the necessity to manage massive amounts of unstructured and semi-structured data [HELD11]. NoSQL databases introduced flexible data structures such key-value, document, columnar, and graph databases in place of the classic relational databases' inflexible schema. These databases provide high availability, scalability, and effective management of various data kinds.

5. NewSQL Databases

   The performance and scalability issues that traditional relational databases encountered in web-scale contexts led to the development of NewSQL databases [Asl11]. These databases sought to combine the scalability and flexibility of NoSQL databases with the advantages of relational databases, such as ACID compliance. Improved performance, distributed architectures, and horizontal scalability were all features of NewSQL solutions.

1.2.8 Advances in Testing and Security

The techniques of software development have been considerably changed by improvements in testing and security. Through the automation of repetitive operations, the reduction of errors, and the advancement of software quality, automated

testing has completely changed the process [Ngu16]. In order to promote quicker feedback loops and effective release cycles, continuous integration and continuous deployment (CI/CD) approaches combine automated testing with seamless code integration and deployment [Hum10]. Through simulated attacks, penetration testing assists businesses in proactively identifying weaknesses [Eng13]. Security flaws and poor code quality can be found using static and dynamic code analysis techniques [Spi06]. Fuzz testing uses random or mutated inputs to stress-test software and uncover vulnerabilities [SGA07]. Threat modeling makes it possible to systematically identify and reduce potential security threats. [Sho14]. These developments aid in the creation of reliable and secure software systems, increasing resilience and dependability.

1.3 Objective

The main goal of this thesis is to create and put into practice a web application that encourages users to connect with other travelers, discuss their travel experiences, and explore new places.

Chapter 2

Theoretical frame

2.1 HTTP

HTTP (Hypertext Transfer Protocol) is an application layer protocol in the Internet protocol suite model (TCP/IP) for data exchange in a client-server interaction. Messages sent by the client are called requests and the answers they receive from the server are called responses.

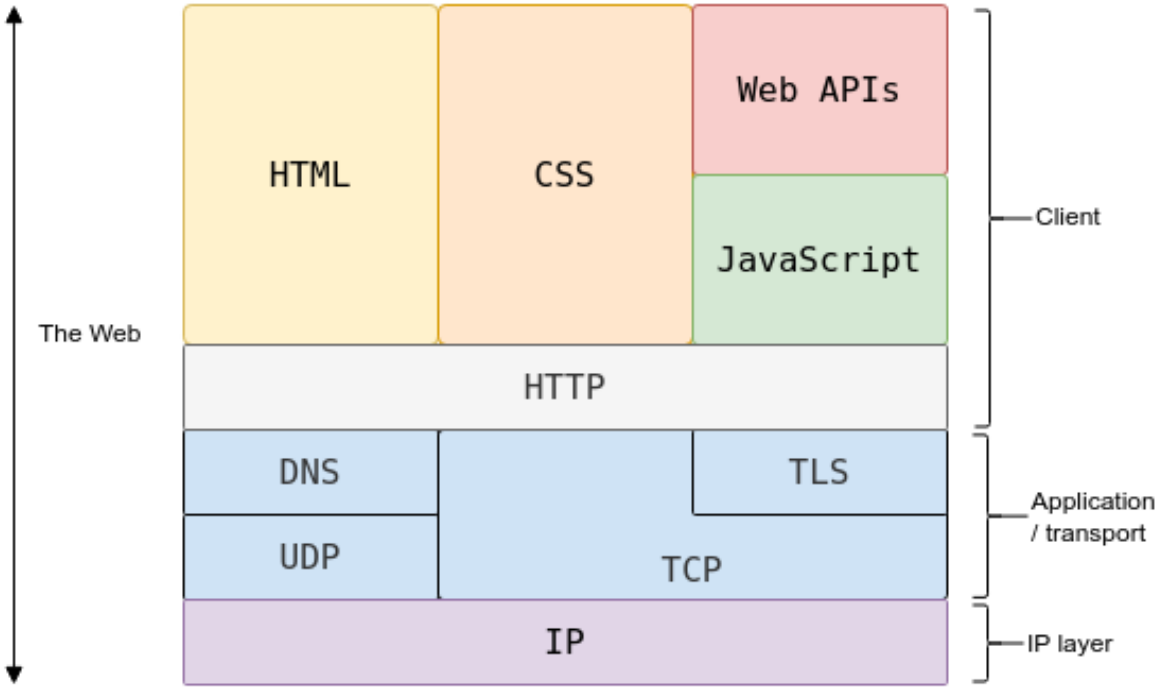

Figure 2.1: The web from the client layer to the IP layer [Moza]

2.2 REST

REST (Representational State Transfer) is an architectural style for designing network-based applications, particularly web services and is most used by HTTP client-server applications. It was introduced by Roy Thomas Fielding in 2000 for his doctoral dissertation. [Fie00]

The architecture is defined by the following constraints:

1. Client-Server - the separation of client and server gives each of them more portability, flexibility and thus leverages the scaling of the applications. This also increases cohesion within each side and decreases coupling between them. Having each of them deal with their own concerns also follows the Single Responsibility Principle of SOLID.

2. Stateless - every client request contains enough information in order for the server to understand and process the request. Only the client stores session state. This simplifies the work of the server.

3. Cache - requiring the implicit or explicit labelling of the response as cacheable or non-cacheable in order to improve of network efficiency

4. Uniform Interface - means having an uniform interface between the client and the server and implies:

    - identification of resources - URI are used to identify resources and those resource are different from the representation sent to the client since a single resource can have multiple representation
    - manipulation of resources through representations - if a client holds a representation of some resource, it has enough information to modify or delete it, provided they have access
    - self-descriptive messages - messages contain enough information to describe how they should be processed (example: specifying the media type)
    - hypermedia as the engine of application state - other resources can be accessed by the hyperlinks provided by the server

5. Layered System - a client can only see the immediate layer they are interacting with. Whether that layer is the final service or an intermediary is known

6. Code-On-Demand (optional) - extending client functionality by downloading and executing code as applets/scripts.

Throughout the application, the JSON representation (visible in Figure 2.2) will be used for client-server interactions.

```json
{
  "id": 1,
  "name": "Linus Torvalds",
  "email": "torvalds@linux-foundation.org"
}
```

Figure 2.2: Example of JSON serialized data

2.3 Middleware

Middleware is code that runs between the incoming HTTP request and the outgoing HTTP response. It is used to process requests and responses, handle specific tasks and make modifications before the request finally reaches the handler outside the middleware pipeline or the response is sent back to the client.

Use cases for middleware

- request processing - inspection, modification or rejection of incoming HTTP request before they reach their final handler. Examples: JSON parsing, handling file uploads, checking specific headers.
- authentication and authorization - examples: verify credentials or enforce role-based access controls
- logging and monitoring - examples: logging requests/response and error tracking
- error handling - it can catch and handle errors that occur during request processing and allows customization of the error messages and the http status codes returned to the client
- rate limiting and throttling - example: limit the number of requests/time in order to prevent abuse
- data transformation - modifying requests and responses by converting formats, adding default values or transforming the data before they reaches the final handler or are returned to the client

2.4 The Software Development Life-cycle

The Software Development Life-cycle (SDLC) is the process that development teams go through in order to create a software product, from gathering the client's requirements till the end of their collaboration for that software's purpose.

The steps involved in SDLC may differ from team to team but a common denominator can be found among the following:

1. Planning - involves analyzing the cost-benefit, scheduling, resource estimation and resource allocation. Requirements are gathered from decision makers involved in the project, experts and customers.

2. Design - By analyzing the requirements gather in the step before and taking into consider the resources available, the team comes with the optimal solution that can fullfil the requirements. This comes in the form of diagrams, schemas, design documents and mockups.

3. Implementation - the team "converts" the design results into code

4. Testing - the team performs various types of testing, identifies and fixes bugs and makes sure the application meets the specified requirements

5. Deployment - packaging, environment configuration and installation are done on the client's environment (usually called *production*)

6. Maintenance - post-deployment monitoring, patching and improvement based on how well the software behaves and the experiences of its users. In this phase, one or more features of the software may go through the whole SDLC phase again.

There are some known models that implement SDLC:

- Waterfall - the design goes from one phase down to next, hence the name. Each phase depends on the result of the previous phase, sequentially, like non-threaded code. The advantage of it is that it has well-defined stages with clear outcomes for each phase. But the trade-off is that you're not meant to back on previously completed phases and make changes, which makes it inflexible. This model is suited for small projects.

- Iterative - the software is built in iterations of small subsets of the requirements. This makes it easier to identify risks and problems and allows changes to be made at any stage. But, since iterations are short, it requires close collaboration with the client, which might not be always possible. It introduces complexity for managing and integrating all the iterations. Requirement changes

can cause over budget and problems could arise in the system architecture since the set of requirements is not predefined.

- Spiral - it is the combination of Waterfall and Spiral models with focus on risk analysis. So, the process goes multiple times through the phases. It is flexible and evaluations at each spiral loop help staying in touch with the goal. The downsides are that it requires complex documentation and management which lead to high costs so it's not fitting for small projects.

- Agile - the development is split into short cycles that go quickly through each SDLC phase resulting in small changes at the end of each cycle. It allows changes at any stage, it provides frequent and early delivery and is continuously improved and refined through feedback from the customer. But it can lead to exaggerated scope changes and the scheduling and costs become less predictable. [Ama]

Chapter 3

Technical frame

3.1 Backend

3.1.1 .NET and ASP.NET Core

.NET (pronounced as ”dot net”) is a free, cross-platform, open-source developer platform for building many kinds of applications including web, desktop, mobile, cloud, IoT. It can run programs written in multiple languages, with C# being the most popular. It relies on a high-performance runtime that is used in production by many high-scale apps. [Mic] It was released in 2016 and its the successor of .NET Framework which dates way back to 2002.

ASP.NET Core is a free, cross-platform, open-source framework for building web apps and services with .NET and C#. It is part of the app stack provided by .NET. Key features are: dependency injection, middleware pipeline, MVC, security.

3.1.2 Entity Framework Core

Entity Framework Core (EF Core) is an open source and cross-platform Object-Relational Mapping (ORM) framework for .NET applications. It is the successor to the original Entity Framework (EF) and has been redesigned to be lightweight and extensible. EF Core allows developers to work with a database using .NET objects, eliminating the need to write most of the data-access code manually. Key concepts:

DbContext is the primary class reponsible for interacting with the database. It manages the entity objects during runtime, tracks changes, handles the database interaction and allows customization of the ORM behavior.

DbSet<T> class represents a collection of entities of a specific type (‘T‘). Each DbSet in a DbContext corresponds to a table in the database and entity instance corresponds to a row in that table.

Entities are classes that map to tables in a relational database. Each property

in an entity class corresponds to a column in the table (with certain exceptions, since EF allows defining a more complex behavior, as stated in `https://learn.microsoft.com/en-us/ef/core/modeling/backing-field`) and each instance of the class corresponds to a row.

LINQ (Language Integrated Query) is used to query the database in a types-safe and object-oriented manner. It allows developers to write queries against the DbContext using standard C# syntax:

```
1 var products = context.Products
2                       .Where(p ⇒ p.Price > 50)
3                       .ToList();
```

Migrations is a feature that helps managing database schema changes over the database's life-cycle. They allow you to incrementally apply changes to the schema, such as adding new tables, modifying columns, creating relationships between tables, etc, while preserving exitsting data.

Change tracking: EF Core automatically keeps track of changes made to entities during the lifetime of a DbContext instances. When *SaveChanges()* is called, EF Core determines what changes have been made and generates the appropriate SQL commands to update the database.

Relationships: EF Core allows you to define relationships between entities, such as one-to-one, one-to-many, and many-to-many relationships. These relationships are represented using navigation properties and are enforced through foreign keys in the database.

3.2 Frontend

3.2.1 React

React is an open-source front-end JavaScript library released in 2013 by Meta (formerly Facebook). Back in 2011, the developers at Facebook started to face some issues with code maintenance. Which was caused by the increase in app features combined with cascading updates. The solution they arrived at is React. [Fer] React does things by breaking down the UI into reusable components. This component-based architecture allows to build complex user interfaces by composing small, isolated pieces of code that manage their own state and logic. Key concepts:

JSX (**JavaScript XML**, formally **JavaScript Syntax eXtension**) is an XML-like extension to the JavaScript language syntax. [Facc] It just a syntatic sugar that is transpiled into nested JavaScript function calls structurally similar to the original JSX.

JSX lets you write HTML-like markup inside a JavaScript file. Inside React, it is used to define components.

Components are the UI building blocks in React. They are self-contained pieces of UI that can be reused throughout the application and are defined using JSX. They can be either functional (as the return of a function) or class-based (created by extending React.Component). Throughout the app, only functional components are used. Example of a functional component:

```
1 function Greeting() {
2   return <h1>Hello, world!</h1>;
3 }
```

Props (short for properties) are read-only data passed from a parent component to a child component. Props allow components to be dynamic and reusable by passing different data into them.

Hooks are function provided by the React API that provide features like ”remembering” information like user input (useState), receiving information from a distant parent without passing it as props (useContext), holding information that isn't used for rendering (useRef), connecting to and synchronizing with external systems (useEffect). [Faca]

State represents the internal writable data insides a component. State is declared using the **useState** hook which returns 2 the value of the state and a function to set that value. The value can be used directly in the JSX or to compute some other state. The setter should only be called inside other hooks or event handlers and not during rendering (i.e. directly inside functional components, which should be pure functions).

Virtual DOM In case of big pages, re-rendering can become an expensive operation. That's why react uses a virtual DOM, which is a copy of the real DOM. When the state of a component changes, a snapshot of its current virtual DOM is saved, then the virtual DOM is modified and compared with the snapshot and only then, the real DOM is changed based on the difference between the initial and the modified virtual DOM.

3.2.2 TypeScript

TypeScript is a strongly typed, compiled programming language that builds on JavaScript, one of the most widely used languages for web development. Developed and maintained by Microsoft, TypeScript introduces static typing to JavaScript,

along with other features aimed at improving developer productivity and code maintainability. It has rapidly gained popularity in the web development community and is commonly used in conjunction with frameworks such as Angular, React, and Vue.js.

TypeScript is a superset of JavaScript, meaning any valid JavaScript code is also valid TypeScript code. TypeScript's main feature is its optional static typing, which allows developers to define the types of variables, function parameters, and function return values. This helps catch errors at compile time rather than at runtime, leading to more robust and maintainable code.

TypeScript is transpiled into plain JavaScript, which can run in any environment that supports JavaScript, including web browsers, Node.js, and various JavaScript engines. The TypeScript compiler checks the code for type errors and compiles it into JavaScript that is compatible with the target environment.

TypeScript has type inference, which means the compiler can deduce the types of variables based on the values they are assigned. It also introduces interfaces, type aliases, access modifiers, generics and ES6 modules.

Using TypeScript comes with the advantage of strong typing, but the disadvantages are: the learning curve, the compilation step - converting the TypeScript code to JavaScript, which adds an extra layer of complexity to the development workflow and the dependency management.

3.2.3 Redux

Redux is an open-source JavaScript library for managing and centralizing application state. In React, centralized state management can be achieved to some degree using the *useContext* feature. But, for medium to large-sized codebases with large amounts of application state and frequent and complex update logic, that might become too complex to maintain. Redux uses a single source of truth which contains read-only state that can only be modified through pure functions called reducers. The core components of Redux are the following:

- Store - is the central object that holds the application's state. It is created using the *createStore* function provided by Redux. The store provides methods to access the state, dispatch actions, and subscribe to changes.
- State - is a single JavaScript object that represents the entire application's data. It can only be updated by dispatching actions and processing those actions in reducers.
- Actions - are plain JavaScript objects that describe what happened in the application. Each action has a type property (a string constant) and can optionally

have a payload property that carries additional data. Actions are dispatched to the store to trigger state changes.

- Reducers - are pure functions that take the current state and an action as inputs and return a new state. Reducers specify how the state should change in response to an action. Since reducers are pure functions, they do not modify the original state but return a new state object.

When a value in the state affects what is displayed to the user (example: the number of items in a shopping cart), that means that state defines the UI. When a user interacts with the page, by clicking a button or inputting text, an action is triggered (example: adding an item to the shopping cart). This action is dispatched to a reducer (example: the reducer receives an action of type addItemToCart with the payload being the id of the item). The reducers then updates the store (example: by adding the new item to the state that contains the shopping cart's items). And, since store contains the state, the changes are reflected everywhere the state is used (example: the count on the shopping cart increased by one and if you go to the checkout page, you will also see item you've just added). This can be seen in Figure 3.1

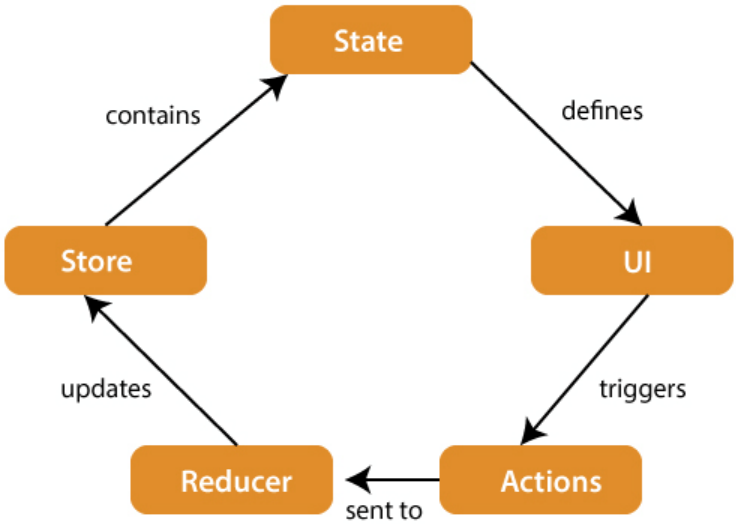

Figure 3.1: Redux flow [Facb]

3.2.4 Material UI

Material UI (often abbreviated as MUI) is an open-source library that provides React components implementing Google's Material Design principles. Material Design is

a design language developed by Google that emphasizes simplicity, consistency, and usability. Material-UI offers a comprehensive set of components and tools that help developers build aesthetically pleasing, responsive, and accessible user interfaces for web applications. Material UI provides various components like custom inputs, custom buttons, avatars, cards, dialogs, app bars, floating action buttons. Besides, it provides a powerful theming system that allows the customization of the appearance of components globally across the application, responsiveness through breakpoints and grids. It also includes a comprehensive set of Material Design icons that can be easily integrated into components. Icons are provided as React components and can be customized in terms of size, color, and other properties.

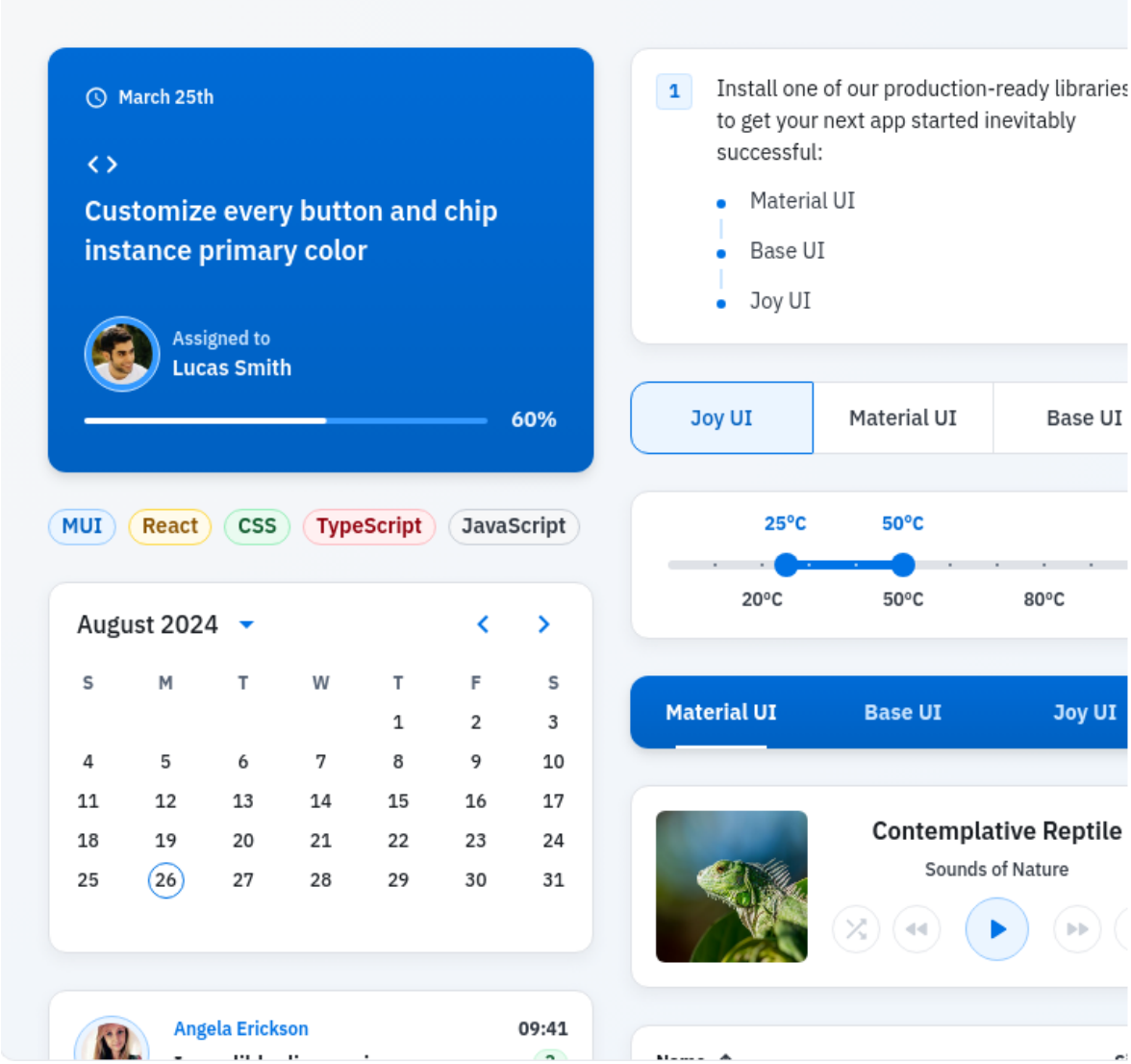

Figure 3.2: An example of what can be achieved with Material UI [Mat]

Chapter 4

Application

4.1 Analysis

4.1.1 Functional requirements

Throughout the application, the user can do the following:

- Unauthenticated user
  - Create an account
  - Login into the application
  - Reset their account's password
- Regular user
  - Log out of the application
  - Change between light and dark theme
  - Visualize, search, sort and filter attractions
  - Add a new attraction or edit a created one
  - React to an attraction with like or dislike
  - Save an attraction to a new or existing collection
  - Share an attraction to Threads, Twitter, Email or copy its link
  - View an attraction's details on its page
  - Comment on an attraction's page
  - View their own and other users' profile
  - Add or change their profile photo and description
  - Send, accept and decline friend requests and unfriend current friends

  - View their own and other users' friends
  - View their own sent or received friend request
  - View the list of their created attractions
  - View the list of their collections
  - Reorder their list of collections
  - Add a new collection or edit/delete a created one
  - View the details of a collection
  - Reorder attractions within a collection
  - Delete an attraction from the collection
  - Set the picture of an attraction as the collection's picture.
- Admin
  - Anything a regular user can do
  - View the list of attraction types and how many attractions are using them
  - Add or rename attraction types
  - Delete unused attraction types

4.1.2 Use cases

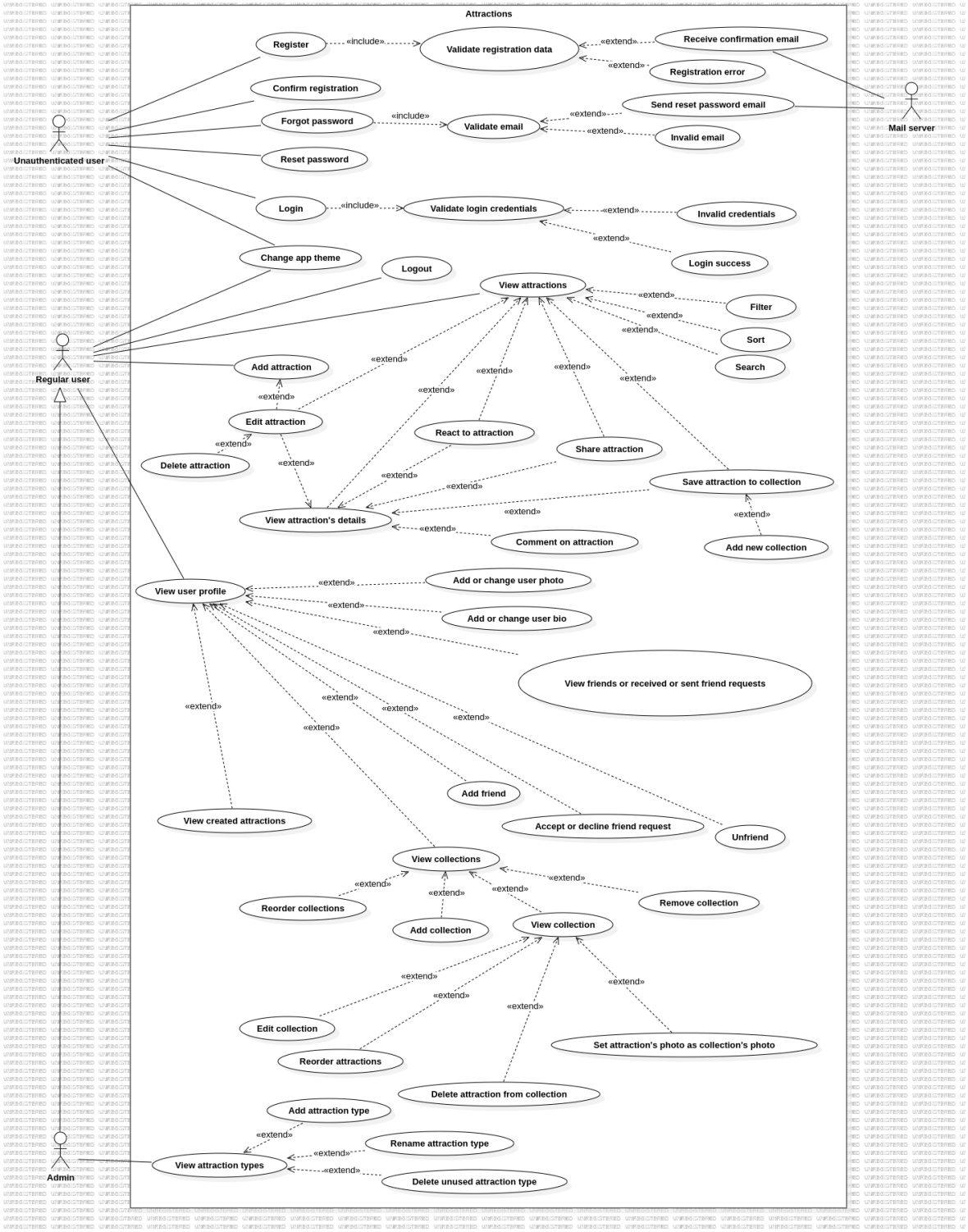

Figure 4.1: Use case diagram

<table>
<tr><td colspan="3">Use case: Register</td></tr>
<tr><td>Actors</td><td colspan="2">Regular User</td></tr>
<tr><td>Standard process</td><td colspan="2"><table>
<tr><td>No.</td><td>User action</td><td>System response</td></tr>
<tr><td>1</td><td>User enters username, email and password then clicks the ”Register” button</td><td></td></tr>
<tr><td>2</td><td></td><td>The systems validates the data, creates the account, sends the confirmation email and then displays a notification</td></tr>
</table></td></tr>
<tr><td>Alternative process</td><td colspan="2"><table>
<tr><td>No.</td><td>User action</td><td>System response</td></tr>
<tr><td>1</td><td>User enters username, email and password then clicks the ”Register” button</td><td></td></tr>
<tr><td>2</td><td></td><td>The systems validates the data, finds validation problems and shows the user the problems with each registration fieldd</td></tr>
</table></td></tr>
<tr><td>Preconditions</td><td colspan="2">The user must be on the register page</td></tr>
<tr><td>Postconditons</td><td colspan="2">If the registration is successful, the user will receive a confirmation email</td></tr>
</table>

4.2 Architecture

The application is structured on the client - server model. The client is a web application created using React and the server is a Web API created using the ASP.NET Core subset of the .NET Framework. For persistent storage the database used is SQLite. External services are used for email sending and photo upload.

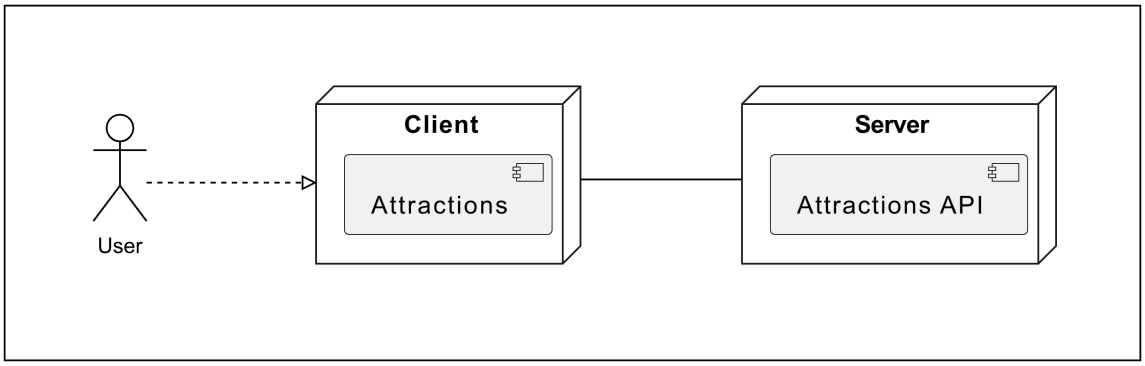

Figure 4.2: Application architecture

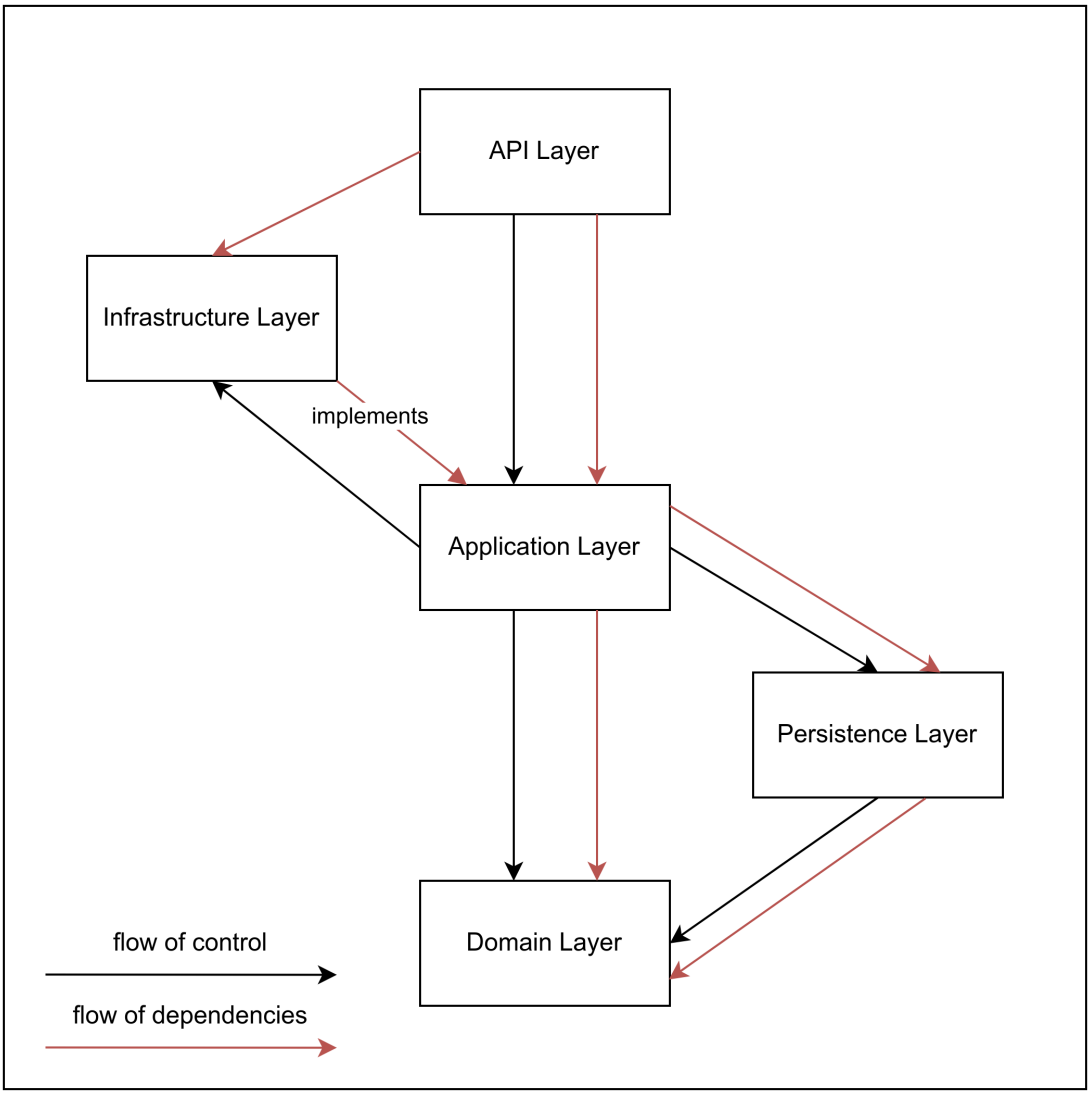

Figure 4.3: Backend layers

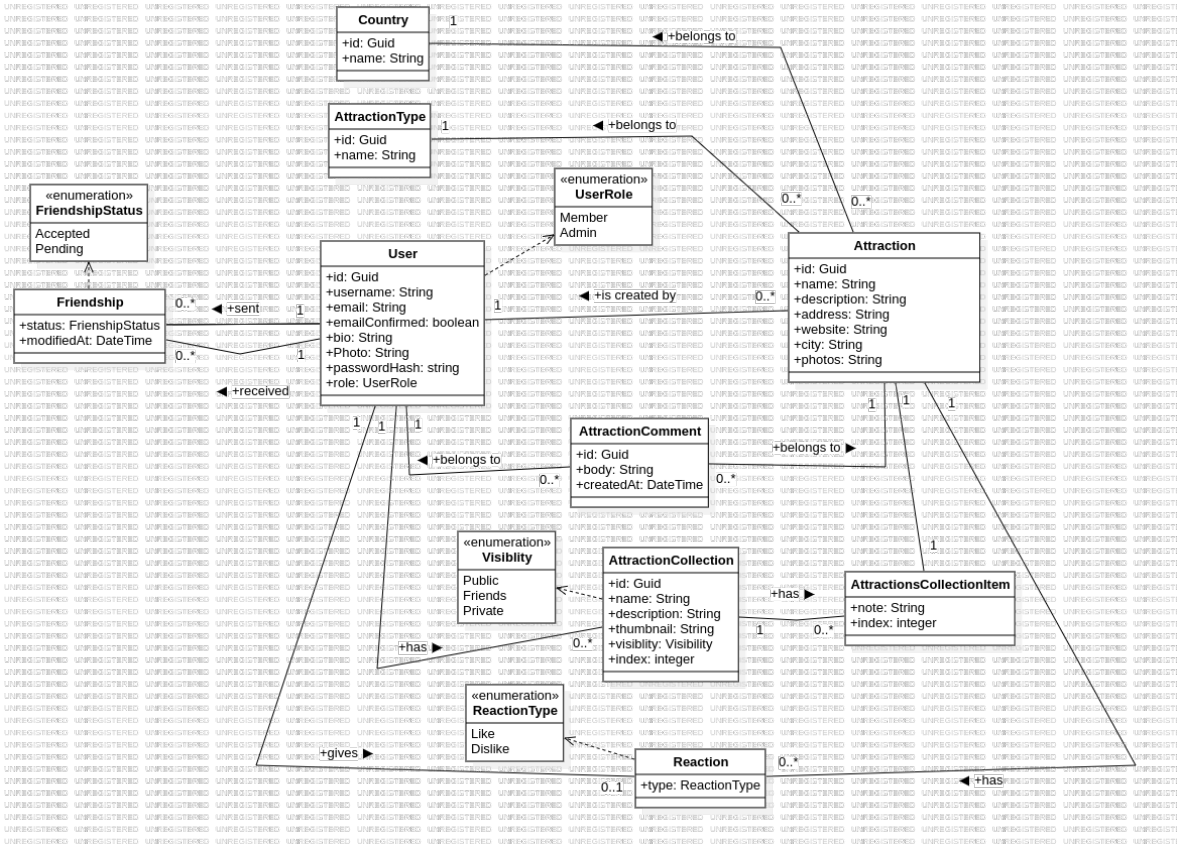

Figure 4.4: Domain class diagram (*associations should be aggregations)

4.2.1 Diagrams

Domain class diagram

Database diagram

Since the database is generated by the Entity Framework, its structure is similar to the domain:

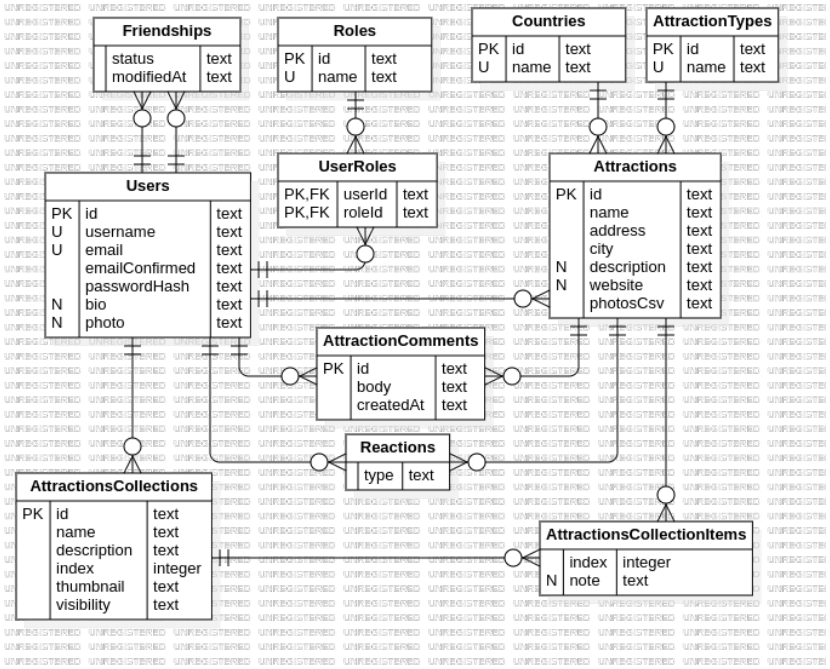

Figure 4.5: Database diagram

Business logic class diagram

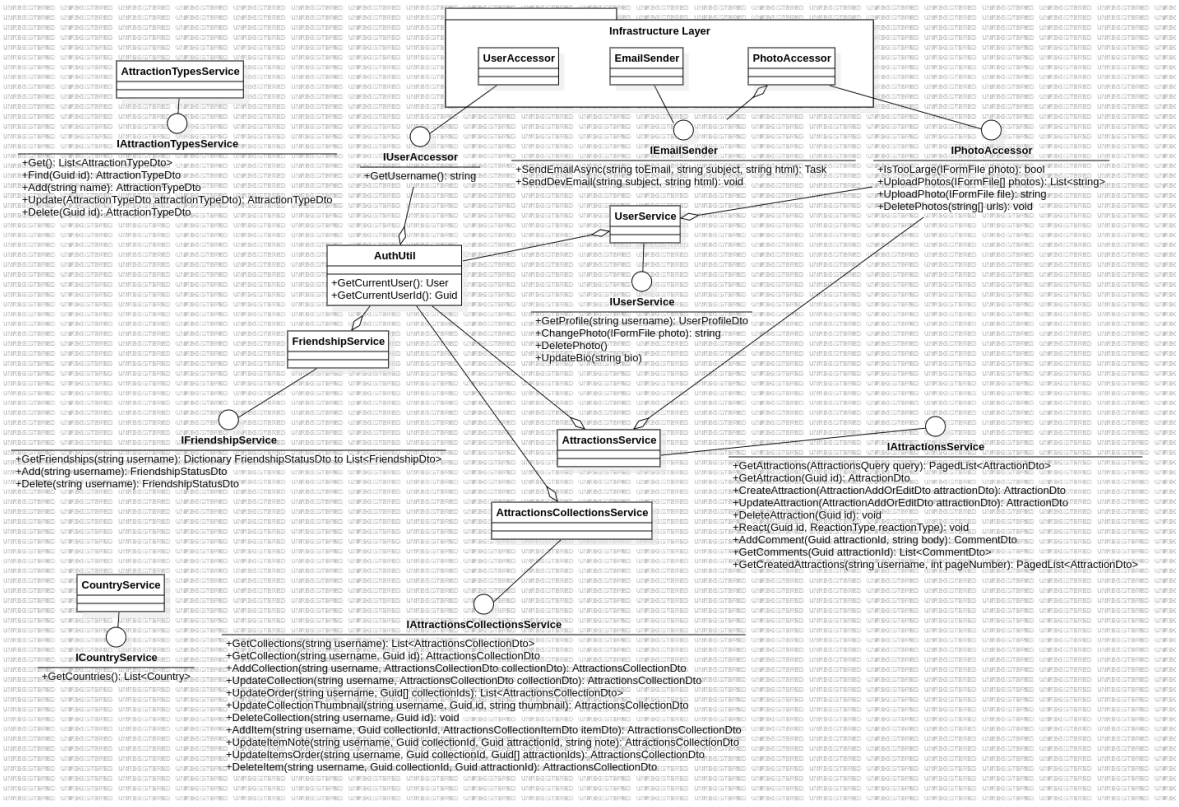

Figure 4.6: Business logic class diagram

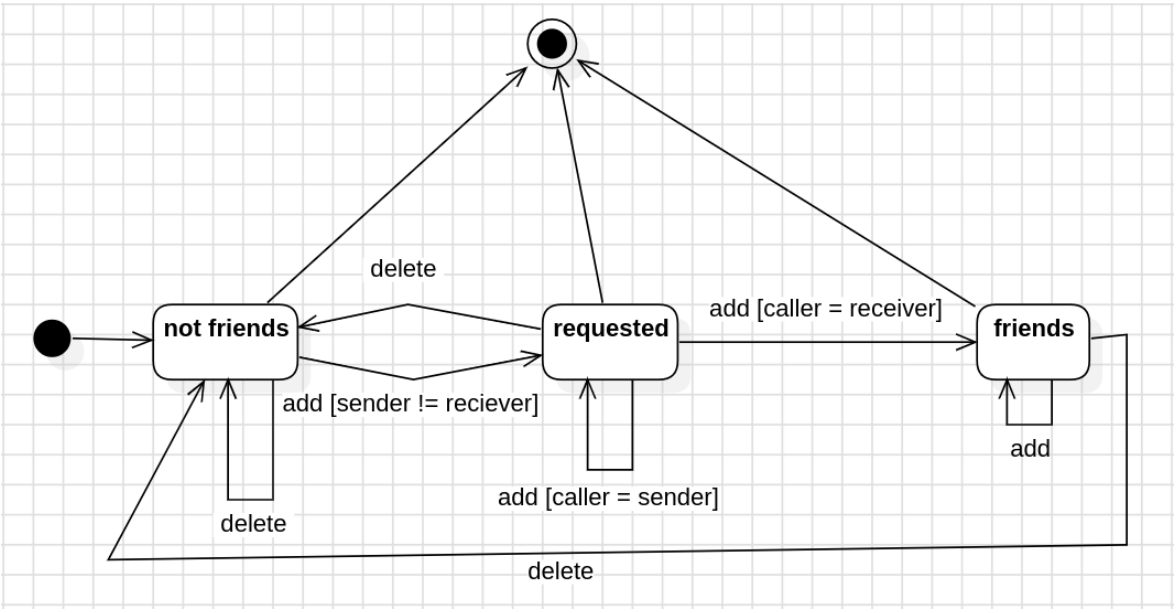

Figure 4.7: Friendship statechart diagram

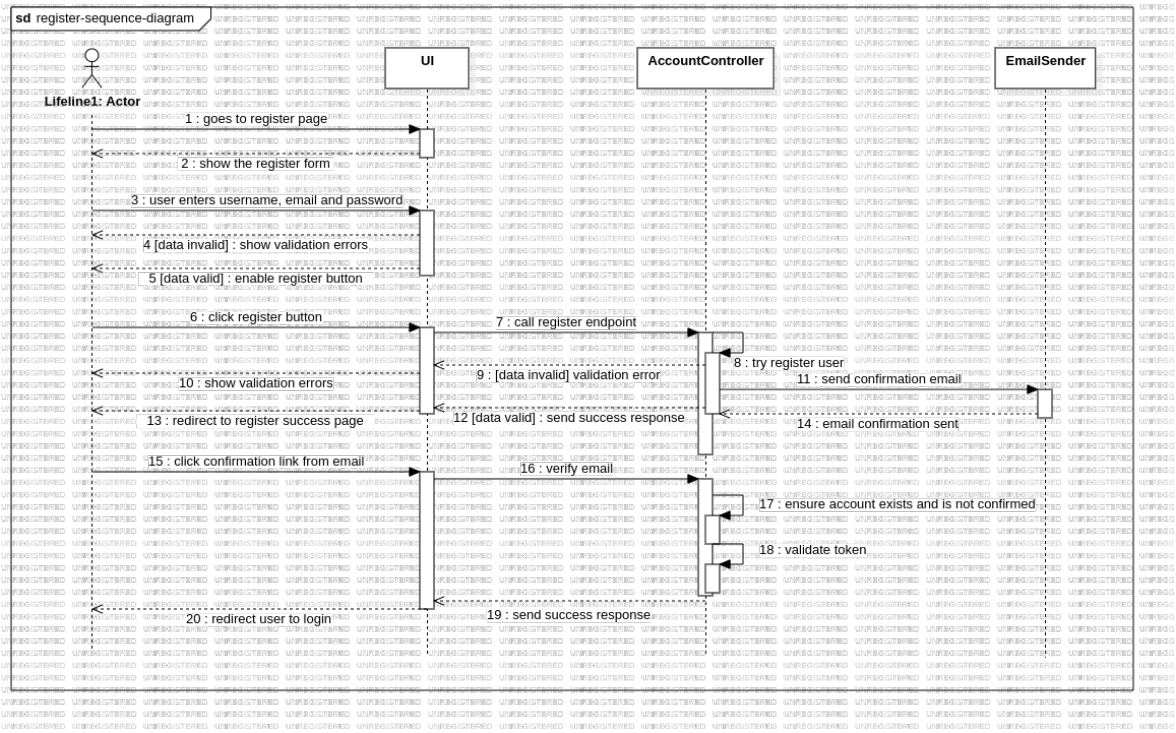

Figure 4.8: Register sequence diagram

4.3 Implementation

4.3.1 ORM Mapping

Entity Framework was used to map the domain classes to the database tables. In order to achieve that, the domain classes have to be used as type parameters for fields of type *DbSet* inside a class that inherits from the framework's class *DbContext*. In this case the class is *IdentityDbContext* because it offers user management features and it is also a descendant of the *DbContext* class. This *IdentityDbContext* class takes as type parameters a class used for mapping the user, a class used for mapping the user role and the type of the primary key used for those two.

```
35 usages  Patrulescu-Ronald-Sandrino
10  public class DataContext(DbContextOptions options) : IdentityDbContext<User, UserRole, Guid>(options)
11  {
        7 usages
12      public DbSet<AttractionType> AttractionTypes { get; init; }
        11 usages
13      public DbSet<Attraction> Attractions { get; init; }
        5 usages
14      public DbSet<Country> Countries { get; init; }
        6 usages
15      public DbSet<Reaction> Reactions { get; init; }
        4 usages
16      public DbSet<AttractionComment> AttractionComments { get; init; }
        9 usages
17      public DbSet<AttractionsCollection> AttractionsCollections { get; init; }
        1 usage
18      public DbSet<AttractionsCollectionItem> AttractionsCollectionsItems { get; init; }
        6 usages
19      public DbSet<Friendship> Friendships { get; init; }
20
```

Figure 4.9: *DataContext* class' fields

The to be generated tables can be further configured by overriding the *OnModelCreating* method of the *DataContext* class. There you can define more complex primary keys, foreign keys, delete behaviors, unique indexes, check constraints, conversions, etc.

```
115         builder.Entity<AttractionsCollectionItem>(b :EntityTypeBuilder<AttractionsCollectionItem> =>
116         {
117             b.HasOne(i :AttractionsCollectionItem => i.Collection) // ReferenceNavigationBuilder<AttractionsCollectionItem,...>
118                 .WithMany(c :AttractionsCollection => c.CollectionItems)
119                 .HasForeignKey(c :AttractionsCollectionItem => c.CollectionId)
120                 .OnDelete(DeleteBehavior.Cascade);
121
122             b.HasOne(i :AttractionsCollectionItem => i.Attraction) // ReferenceNavigationBuilder<AttractionsCollectionItem,...>
123                 .WithMany()
124                 .HasForeignKey(i :AttractionsCollectionItem => i.AttractionId);
125
126             b.HasKey(i :AttractionsCollectionItem => new { i.CollectionId, i.AttractionId });
127
128             b.HasIndex(ci :AttractionsCollectionItem => new { ci.CollectionId, ci.AttractionId }).IsUnique();
129             b.HasIndex(ci :AttractionsCollectionItem => new { ci.CollectionId, ci.AttractionId, ci.Index }).IsUnique();
130         });
131     }
```

Figure 4.10: Example of configuring the database tables generation by overriding the *OnModelCreating* method

Furthermore, the defined *DataContext* class has to be registered to the application services by using the *AddDbContext* method. This also registers it for dependency injection so it can be used inside the service classes. This is also where the database provider is set up, which requires a method for registering it (which is usually provided by the package used for the database provider), *UseSqlite* in this case, and a connection string.

```
31          services.AddDbContext<DataContext>(options =>
32          {
33              options.UseSqlite(configuration.GetConnectionString( name: "DefaultConnection"));
34          });
```

Figure 4.11: Registering the **DataContext** class

4.3.2 Automapper

Automapper is an object-object mapper. It is used in this app in order to do conversions between domain classes and DTO classes. Mappings can be configured using profiles, which are classes that inherit from Automapper's Profile class. The profile is then registered to the DI container of application services:

```
47          services.AddAutoMapper(typeof(MappingProfiles).Assembly);
```

Figure 4.12: Registering Automapper to the application's DI container

Mapping for fields with the same name happens implicitly. For the rest of them, we can define custom mappings or even ignore them, as seen in Figure 4.13, at lines 15 and 21 respectively.

```
1 usage  Patrulescu-Ronald-Sandrino
public class MappingProfiles : Profile
{
    Patrulescu-Ronald-Sandrino
    public MappingProfiles()
    {
        CreateMap<Attraction, Attraction>();
        Guid? currentUserId = null;
        CreateMap<Attraction, AttractionDto>()
            .ForMember(d => d.Country, o => o.MapFrom(s => s.Country.Name))
            .ForMember(d => d.AttractionType, o => o.MapFrom(s => s.AttractionType.Name))
            .ForMember(d => d.Reaction,
                o => o.MapFrom(s =>
                    s.Reactions.Where(r => r.UserId == currentUserId).Select(r => r.Type).FirstOrDefault()));
        CreateMap<AttractionAddOrEditDto, Attraction>()
            .ForMember(d => d.Photos, o => o.Ignore());
        CreateMap<AttractionDto, Attraction>();
        CreateMap<AttractionType, AttractionType>();
        CreateMap<AttractionComment, CommentDto>()
            .ForMember(d => d.AuthorUsername, o => o.MapFrom(s => s.Author.UserName))
            .ForMember(d => d.AuthorPhoto, o => o.MapFrom(s => s.Author.Photo));
```

Figure 4.13: Example of Automapper profile configuration

```
110         var comment = new AttractionComment
111         {
112             Body = body,
113             Author = user,
114             Attraction = attraction,
115         };
116
117         attraction.Comments.Add(comment);
118
119         var success bool = await context.SaveChangesAsync() > 0;
120         if (!success) throw new Exception("Failed to add comment");
121
122         return mapper.Map<CommentDto>(comment);
```

Figure 4.14: Example of Automapper usage (line 122)

4.3.3 Reading from and writing to the database

Reading is done by directly accessing the *DbSet* fields of the *DataContext* class. For example, in Figure 4.15 at line 58 the contents of table of attraction types are accessed by the AttractionTypes field on the variable context, which is an instance of *DataContext* injected into the service. The contents can be further processed on the

database side by using non-terminal operations, methods that return an instance of *IQueryable*, like Where, Include (which does join) and *ProjectTo*. Then, when awaiting the call to *ToListAsync* the processed query is performed on the database.

```
56      private async Task<List<AttractionTypeDto>> Get(Guid? id)
57      {
58          var attractionTypes:List<AttractionTypeDto> = await context.AttractionTypes.Where(at => !id.HasValue || at.Id == id)
59              .Include( navigationPropertyPath: at => at.Attractions) // IIncludableQueryable<AttractionType,List<...>>
60              .ProjectTo<AttractionTypeDto>(mapper.ConfigurationProvider).ToListAsync(); // Task<List<...>>
61          return attractionTypes;
62      }
63
```

Figure 4.15: Reading from the database

Writing to the database is done in 2 steps: 1. calling *Add*/*Remove* (and derivatives like *AddAsync* or *RemoveRange*) on either the *DbSet* field or the DataContext instance, or by directly updating a domain class instance managed by the DataContext and which comes from a read operation and 2. by calling *SaveChanges* on the *DataContext* instances. An add example can be seen in Figure 4.14 and an update example can be seen in Figure 4.16

```
37      public async Task<AttractionTypeDto> Update(AttractionTypeDto attractionTypeDto)
38      {
39          var attractionType = await FindInner(attractionTypeDto.Id);
40          attractionType.Name = attractionTypeDto.Name;
41          await context.SaveChangesAsync();
42          return mapper.Map<AttractionTypeDto>(attractionType);
43      }
44
        0+1 usages  Patrulescu-Ronald-Sandrino
45      public async Task<AttractionTypeDto> Delete(Guid id){...}
55
        2 usages  Patrulescu-Ronald-Sandrino
56      private async Task<List<AttractionTypeDto>> Get(Guid? id){...}
63
        2 usages  Patrulescu-Ronald-Sandrino
64      private async Task<AttractionType> FindInner(Guid id)
65      {
66          return await context.AttractionTypes // DbSet<AttractionType>
67              .Where(at => at.Id == id) // IQueryable<AttractionType>
68              .Include( navigationPropertyPath: at => at.Attractions) // IIncludableQueryable<AttractionT
69              .FirstOrDefaultAsync() ?? throw new NotFoundException();
70      }
71  }
```

Figure 4.16: Example of database update operation

4.3.4 URL Mapping

Each URL path is prefixed by the name of the controller. This is achieved by having all the controllers inherit from BaseApiController, which defines the path prefix using the Route annotation (Figure 4.17). The rest of the URL path is defined by the argument of the annotations used for declaring the HTTP methods used (Figure 4.18). It can be observed that the paths can be parameterized.

```
4   namespace API.Controllers;
5
6   [ApiController]
7   [Route( template: Route)]
8   [Produces( contentType: MediaTypeNames.Application.Json)]
        10 usages  10 inheritors  Patrulescu-Ronald-Sandrino
9   public class BaseApiController : ControllerBase
10  {
11      protected const string RoutePrefix = "api";
12      protected const string Route = $"{RoutePrefix}/[controller]";
13  }
14
```

Figure 4.17: How the name of the controller is added to the URL

```
54      [HttpGet( template: "form-data/{id:guid?}")]
            2 usages  Patrulescu-Ronald-Sandrino
55      public async Task<ActionResult<AttractionFormData>> GetAttractionFormData(Guid? id){...}
70
71
72      [HttpPut( template: "{id:guid}/react")]
            Patrulescu-Ronald-Sandrino
73      public async Task<ActionResult> React(Guid id, ReactionType reactionType){...}
78
79      [HttpGet( template: "{username}")]
            Patrulescu-Ronald-Sandrino
80      public async Task<List<AttractionDto>> GetCreatedAttractions(string username, [FromQuery] int pageNumber){...}
86  }
87
```

Figure 4.18: Defining endpoints

4.3.5 Dependency Injection

Dependency injection is achieved by specifying the needed classes as constructor parameters (4.19) and then registering those classes when the application is created (4.20). Here, they are registered using the *AddScoped* method, which registers their lifetime per HTTP request.

```
21  public class AttractionsService(
22      DataContext context,
23      IMapper mapper,
24      AuthUtil authUtil,
25      IPhotoAccessor photoAccessor)
26      : IAttractionsService
27  {
```

Figure 4.19: Injection of dependencies using the primary constructor

```
36  services.AddScoped<IAttractionTypesService, AttractionTypesService>();
37  services.AddScoped<IAttractionsService, AttractionsService>();
38  services.AddScoped<ICountryService, CountryService>();
39  services.AddScoped<IUserService, UserService>();
40  services.AddScoped<IAttractionsCollectionsService, AttractionsCollectionsService>();
41  services.AddScoped<IUserAccessor, UserAccessor>();
42  services.AddScoped<IFriendshipService, FriendshipService>();
43  services.AddScoped<IAdminService, AdminService>();
44  services.AddScoped<IPhotoAccessor, PhotoAccessor>();
45  services.AddScoped<IEmailSender, EmailSender>();
46  services.AddScoped<AuthUtil>();
```

Figure 4.20: Registration of dependencies

4.3.6 Error handling

When the Web API is created (in the entry point of the application - the *Program.cs* file) there are 2 things that are done. The first one is the registration of the services (Figure 4.21 lines 18 - 28) and the second is the configuration of the HTTP request pipeline (Figure 4.21 lines 33 - ...). The order in which the middlewares are configured is the order in which they are run. And, the reason the exception handling middleware is the first in the pipeline (Figure 4.21 line 33) is to catch any exception that occurs later in the pipeline. The purpose of the exception middleware is to have a single piece of code that handles what response is given, depending on the exception, to the initiator of the HTTP request. In this application, it changes the status code depending on the exception (Figure 4.22 lines 24 - 33), writes the validation errors to the body of the response (Figure 4.22) line 33) and adds the stacktrace to the response only if the app is running in development mode.

```
13    var builder = WebApplication.CreateBuilder(args);
14    var configuration = builder.Configuration;
15
16    // Add services to the container.
17
18    builder.Services.AddControllers( configure: options =>
19    {...})
26        .AddJsonOptions(options => options.JsonSerialize
27    builder.Services.AddApplicationServices(configuratio
28    builder.Services.AddIdentityServices(configuration,
29
30    var app :WebApplication = builder.Build();
31
32    // Configure the HTTP request pipeline.
33    app.UseMiddleware<ExceptionMiddleware>();
```

Figure 4.21: Registration of the exception handling middleware

```
8   public class ExceptionMiddleware(RequestDelegate next, ILogger<ExceptionMiddleware> logger, IHostEnvironment env)
9   {
10      private static readonly JsonSerializerOptions JsonSerializerOptions =
11          new() { PropertyNamingPolicy = JsonNamingPolicy.CamelCase };
12
        Patrulescu-Ronald-Sandrino
13      public async Task InvokeAsync(HttpContext context)
14      {
15          try
16          {
17              await next(context);
18          }
19          catch (Exception e)
20          {
21              context.Response.ContentType = MediaTypeNames.Application.Json;
22              var response = new ProblemDetails { Title = e.Message };
23
24              switch (e)
25              {
26                  case ForbiddenException:
27                      response.Status = StatusCodes.Status403Forbidden;
28                      break;
29                  case NotFoundException:
30                      response.Status = StatusCodes.Status404NotFound;
31                      break;
32                  case ValidationException ve:
33                      response.Status = StatusCodes.Status422UnprocessableEntity;
34                      if (ve.Errors.Count > 0) response.Extensions.Add("errors", ve.Errors);
35                      break;
36                  default:
37                  {
```

Figure 4.22: Middleware for exception handling

Chapter 5

Conclusions and future work

5.1 Conclusions

ASP.NET Core offers a reliable and easy choice for Web API application development. To further solidify this, the minimal API that can be created is a 4-lines long code (Figure 5.1). And, by adding the Entity Framework into the mix to take care of the database side, all that remains is the business logic.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Figure 5.1: Minimal API with ASP.NET Core [Pus]

Since React is an unopinionated library, it leaves room for a high degree of customization. But that is not without cost, since it could lead to having to reinvent the wheel and/or ending up with tons of dependencies. It's like the space-time trade-off, the best choice depends on the individuality of each project.

5.2 Future work

5.2.1 Internationalization (i18n) and localization (l10n)

Internationalization (i18n) is the provisioning of the application with different languages. Internationalizing the application can expand its the reach to a global audience. Besides, since attractions could be all over the world, having the ability to

switch between languages creates an easy way for comparison, which is handy for language learners.

Localization is the adaptation of the application to specific locales. Specifically, referring to date and time formats; number formats; symbols, icons, and colors; Right-to-Left (RTL) Language Support. This too increases the regional reach of the application.

5.2.2 Accessibility (A11y)

Accessibility means making the web application usable by as many people as possible, including those with disabilities. This increases the potential user base of the application.

5.2.3 Responsive Design

This means making the app usable on various devices, like phones, tablets and even smart TVs. Currently the app is the design for PC/laptop screen sizes only.

5.2.4 Animations

Adding animations to the application will improve the smoothness of the application and the user experience.

5.2.5 Visibility for attractions

Currently, only attractions collections allow setting the visibility. By extending it to attractions, it can improve the user experience by increasing the user's control of what parts of their information is publicly available. And, you can even extend the app's scope beyond the standard meaning of attractions. For example you could add your secret favorite place, even if it's not what you would normally call an attraction.

5.2.6 Shared collections

Are you planning a multi-destination trip with your friends/family and want to put all the suggestions or even the final itinerary in one shared place? That's where shared collections would come in.

5.2.7 Personalized recommendations

All the user interactions produce data: what attractions the user interacts with and how much time they spend on those attractions? what about their friends? is there a pattern to all the collections they made or to the attractions they created? etc. With collected data on such questions, an algorithm could be made that would give personalized recommendations to the users.

5.2.8 Abstracting the usage of the the UI design library

Unlike the other ideas, which have in mind the user experience (UX), this is about the developer experience. The UI design library in this application (Material UI) is used as is, by directly using the components provided. But those components could be abstracted into other components, put together in one place, that only expose functionality and no implementation details of the design library. This way, should a need for the change of the UI design library appear, the required work to achieve that would be way more tedious.

5.2.9 Legal and informational pages

Pages like Terms and Conditions, Privacy Policy, and Contact are crucial for regulatory compliance and for for establishing trust and transparency with users.

Bibliography

[Ama] Amazon. What is SDLC (Software Development Lifecycle)? `https://aws.amazon.com/what-is/sdlc/`. Online; accessed 26 aug 2024.

[Asl11] M. Aslett. The newsql movement. *451 Research*, 1(14), 2011.

[BLF99] T. Berners-Lee and M. Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its inventor*. Harper, USA, 1st edition, 1999.

[CK74] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 20(3), 1974.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 1970.

[Dat03] C. J. Date. *An Introduction to Database Systems*. Pearson, USA, 8th edition, 2003.

[Dja] Django. Django documentation. `https://docs.djangoproject.com/en/4.2/`. Online; accessed 15 March 2023.

[Eng13] P. Engebretson. *The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy*. Syngress, USA, 2nd edition, 2013.

[Faca] Facebook. Built-in React Hooks. `https://react.dev/reference/react/hooks`. Online; accessed 26 aug 2024.

[Facb] Facebook. Built-in React Hooks. `https://react.dev/reference/react/hooks`. Online; accessed 26 aug 2024.

[Facc] Facebook. JSX. `https://facebook.github.io/jsx/`. Online; accessed 26 aug 2024.

[Fer] Ferenc Hámori. History of React.js. `https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/`. Online; accessed 26 aug 2024.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[Gar] Garrett, J.J. Ajax: A New Approach to Web Applications. Adaptive Path. `https://web.archive.org/web/20180823223757/https://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/`. Online; accessed 15 March 2023.

[HELD11] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, 2011.

[Hum10] D. Humble, J. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, USA, 1st edition, 2010.

[Lar] Laravel. Laravel Documentation. `https://laravel.com/docs/10.x`. Online; accessed 15 March 2023.

[Mat] Material UI. Material UI Home Page. `https://mui.com/`. Online; accessed 26 aug 2024.

[Mic] Microsoft. Introduction to .NET. `https://learn.microsoft.com/en-us/dotnet/core/introduction`. Online; accessed 26 aug 2024.

[Moza] Mozilla. An overview of HTTP. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview`. Online; accessed 26 aug 2024.

[Mozb] Mozilla Developer Network. CSS: Cascading Style Sheets. `https://developer.mozilla.org/en-US/docs/Web/CSS`. Online; accessed 15 March 2023.

[Mozc] Mozilla Developer Network. HTML: HyperText Markup Language. `https://developer.mozilla.org/en-US/docs/Web/HTML`. Online; accessed 15 March 2023.

[Mozd] Mozilla Developer Network. JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. Online; accessed 15 March 2023.

[Ngu16] T. A. Nguyen. *Automated software testing*. USA, 1st edition, 2016.

[Pus] Pushpendra. How data flow in React-Redux app. `https://blog.knoldus.com/how-data-flow-in-react-redux-app/`. Online; accessed 24 aug 2024.

[Rub] Ruby on Rails. Ruby on Rails Guides. `https://guides.rubyonrails.org/`. Online; accessed 15 March 2023.

[SG14] Shyam Seshadri and Brad Green. *AngularJS: Up and Running: Enhanced Productivity with Structured Web Apps*. O'Reilly Media, USA, 1st edition, 2014.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, paperback edition, 2007.

[Sho14] Adam Shostack. *Threat Modeling: Designing for Security*. Wiley, kindle edition, 2014.

[Spi06] D. Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley Professional, USA, 1st edition, 2006.

[Ste21] Stoyan Stefanov. *React: Up Running*. O'Reilly Media, USA, 2nd edition, 2021.

[US90] Rainer Unland and Gunter Schlageter. Object-oriented database systems: Concepts and perspectives. In *IBM Symposium: Database Systems of the 90s*, Advances in Database Systems, pages 154–197. Springer Berlin Heidelberg, 1990.

[Vue] Vue.js. The Progressive JavaScript Framework. `https://vuejs.org/`. Online; accessed 15 March 2023.