WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

BACHELOR STUDY PROGRAM: Computer Science in English

TIMIŞOARA ACCESS MAP: A WEB-BASED PLATFORM FOR URBAN ACCESSIBILITY INFORMATION

SUPERVISOR:
Asistent Dr. Florin Rosu

GRADUATE:
Popa Sebastian Gabriel

TIMIŞOARA
2025

Abstract

Ensuring fair urban access for individuals with disabilities is critical for inclusive societies. Despite the importance of accessibility, Timișoara, like many cities, lacks a dedicated, community-driven platform that is comprehensive in accessibility information. The lack of such information creates significant barriers to independent navigation and social participation for individuals with diverse needs. This thesis aims to solve this gap by presenting the design, development, and implementation of "Timișoara Access Map," a web-based application designed and developed as a central hub for accessibility information within the city.

"Timișoara Access Map" is built upon a crowdsourcing model, empowering registered users to submit and update data regarding the accessibility features of publicly designated places. The core of the platform is an interactive map, built with Leaflet.js and OpenStreetMap. It allows users to visualize locations and filter them based on specific accessibility criteria such as ramp access, accessible WCs, Braille signage, and audio guidance systems. The system incorporates user authentication, profile management, and mechanisms for submitting detailed location information, including photographic evidence. To maintain data integrity, user submissions are reviewed by an administrator. Furthermore, users can contribute to the location with reviews and ratings, further validating the dataset.

The backend part for "Timișoara Access Map" is developed using the Flask Python microframework [Pal25a]. Flask was chosen for its flexibility and the extensive selection of extensions it offers. For database interactions, Flask-SQLAlchemy was utilized to implement Object-Relational Mapping [Pal25b]. Flask-Migrate [Gri25] was used to manage database schema changes, Flask-Login [Fra25] for user authentication and session management, Flask-WTF [YJ25] for the handling of secure web forms and protection against CSRF, Flask-Babel [Ron] for internationalization, and Flask-Limiter [Sai25] for the implementation of rate limits on application routes. The project's goal is to include a review of existing accessibility mapping solutions, design a robust system architecture, and adhere to web accessibility standards (WCAG) to ensure the platform itself is usable by its target audience. By delivering a user-friendly platform, it will significantly enhance urban navigation and foster a more inclusive environment in Timișoara by bridging the critical information gap on accessibility.

Contents

Chapter 1

Introduction

1.1 Motivation and Context

Ensuring that everyone, including individuals with disabilities, has fair access to urban environments is a key part of creating modern and inclusive societies [Uni06]. How a city manages the information on such manner greatly affect how independently people with disabilities can move around and participate in the social life of the city. Barriers, whether they are physical structures or a gap in information, can seriously limit people's access to work, education, healthcare, and social activities [Uni06]. Timişoara, with its blend of historic areas, newly developed areas, and many cultural events, often you see that many do not really include information on how accessible it is. While the city is modernizing, older parts can be difficult to navigate for some, while newer parts of the city might more readily include accessibility features.

A critical need for anyone navigating the city is reliable and specific details about the accessibility of different places. For example, someone using a wheelchair needs to know if there's a usable ramp or an accessible restroom. A person with a visual impairment might need information on tactile paving or audio guides [KJH24]. When this kind of detailed information isn't easy to find, it creates a major hurdle, often leading to uncertainty, wasted trips, and a feeling of exclusion [The17]. My motivation for this thesis came from wanting to address this information gap in Timişoara. I aimed to use technology to help empower individuals with disabilities and contribute to a more inclusive city.

1.2 Problem Statement

Even though the idea of an accessible city is so important, I have found that here in Timișoara we really lack a solid, single place where people can find detailed, trustworthy information about accessibility. My review of available tools showed that while general mapping applications like Google Maps offer some basic accessibility data, they often do not provide the specific details or local verification that people with particular accessibility needs depend on for confident planning [Goo25]. The information that *is* available can be spread across different places, might be out-of-date, or simply missing for many venues.

Furthermore, a key issue I identified is the lack of an easy way for the local community—especially those who personally experience accessibility challenges—to share their

knowledge and help keep this information updated and accurate [BA19]. This absence of a central, trustworthy platform makes it harder for people with disabilities, to plan their journeys and move around Timișoara independently. This, in turn, can affect their ability to fully take part in the society.

1.3 Proposed Solution: Timişoara Access Map

To tackle the problems I have outlined, this thesis presents the "Timișoara Access Map": a full stack web application I designed and developed to act as a central hub for accessibility information in Timișoara. This platform works using a crowdsourcing approach. This means registered users can submit information about the accessibility of various public places, businesses and events like restaurants, coffee shops, universities and public concerts. They can also help keep existing information up-to-date. My goal with this model was to create a dataset that is both rich in detail and maintained by the community itself.

The main feature of the application is an interactive map, which I built using the Leaflet.js library [Vla25], with OpenStreetMap [Ope25a] providing the base map images. This map shows Timisoara as an whole, and users can search for each location or navigate the map and press directly on the location and get the information they need; furthermore they can filter the locations based on specific accessibility criteria. These filters, such as whether a place has a ramp, an accessible WC, or Braille signage, are based directly on the information collected through the application.

The application includes standard features like user accounts, register and login forms, users can manage their profiles and customize them, and submit new locations with detailed accessibility information including photos. To help ensure the quality of the data, I implemented an administrative review process: new locations submitted by users are checked by an administrator before they appear on the map for everyone. Users can also write reviews and give ratings for locations, which adds another layer of community-validated information.

1.4 Objectives

During the development of "Timișoara Access Map," I set out a series of clear objectives to guide the project and ensure it would result in a useful and usable platform:

- To research existing accessibility mapping platforms, relevant web technologies, and current accessibility standards to understand the state of the art.
- To design a solid system architecture and a detailed database schema that could effectively store information about locations, users, accessibility features, and reviews.
- To implement a secure system for users to register, log in, and manage their profiles, including a feature for them to set their personal accessibility preferences.
- To create an intuitive interface for users to submit information about new locations, allowing them to detail features like ramps, accessible WCs, parking, entrances, Braille signage, audio guides, and staff assistance.
- To build an interactive map using Leaflet.js that displays approved locations and allows users to filter these locations based on the accessibility criteria they select.
- To develop the functionality for users to add to the platform's value by submitting written reviews and star ratings for locations.

- To create an administrative backend with tools for managing user accounts, reviewing and approving (or rejecting) newly submitted locations, and editing existing location information.
- To build the application's backend using Flask, making good use of its features and extensions like Flask-SQLAlchemy, Flask-Login, Flask-Migrate, Flask-WTF, Flask-Babel, and Flask-Limiter.
- To structure the application's code according to good software engineering practices to make it easier to maintain, test, and potentially expand in the future.
- To make a conscious effort to follow fundamental web accessibility principles, using the WCAG standards [W3C25b] as a guide for the frontend design, so that the platform itself is accessible to its target users.
- To carry out functional testing to check that the core features of the application work as intended and meet the project's requirements.

My aim in achieving these objectives was to create a truly helpful resource for the Timișoara community.

1.5 Thesis Structure

This document outlines the research, design, building process, and evaluation of the "Timișoara Access Map" project. I've organized it into the following chapters to present the work in a clear and logical way:

- Chapter 1: Introduction – This first chapter presents why this project is important, the problem it tries to solve, the solution I developed, my main goals, and an overview of how this document is laid out.

- Chapter 2: State of the Art – Here, I looked at what already exists on the market, reviewing existing accessibility maps, exploring the technologies often used to build them, and looking at important accessibility standards and guidelines.

- Chapter 3: Design and Architecture – This chapter explains how I planned the platform. I describe the overall design, how the database is structured, the design of the backend API, and the main ideas behind the user interface and user experience.

- Chapter 4: Implementation – This is where I get into the technical details of how I actually built the application. I'll talk about the development setup, how I put together the backend (server-side) parts, and how I developed the frontend (what users see and interacts with on the website).

- Chapter 5: Testing and Evaluation – In this section, I discuss how I checked that the application works correctly and is usable. I'll cover the tests I ran, what I found, and how I assessed if it meets accessibility standards.

- Chapter 6: Conclusions and Future Work – To finish, I'll summarize what the project accomplished, think about what went well and any limitations, and suggest some ideas for how the "Timișoara Access Map" could be improved or built upon in the future.

- Bibliography: A list of all the sources—like academic papers, websites, and software documentation—that I referred to while working on this thesis.

Chapter 2

State of the Art

Before starting the development of ”Timișoara Access Map,” it was really important for me to understand what's already out there, what worked and what's missing . So, this chapter takes a look at existing work and common practices related to my project. I've reviewed some well-known accessibility mapping platforms, looked into the main technologies that power these kinds of systems, and checked out the relevant accessibility standards and ways of modeling this type of data. This background research made me more aware of the situation and what is needed to be done.

2.1 Digital Accessibility Mapping

Using digital tools to map and share accessibility information has become quite common, which makes sense with all the advances in web tech, smartphones, and open data efforts. These platforms can be pretty different from each other in terms of how big they are, how they get their data, and who they're trying to help.

2.1.1 Global Platforms and Approaches

Looking at the global scale, a few big platforms stand out and offer good examples of what can be done, as well as highlighting some common challenges:

- **Wheelmap.org:** This is a really well-known project from a German non-profit called Sozialhelden e.V. [Soz25]. Wheelmap is a global map where people can mark places as wheelchair accessible (fully, partly, or not at all). It's built on OpenStreetMap (OSM) data. Its success shows how powerful an crowdsourcing model is, but it also points to the difficulty of keeping information super detailed and accurate everywhere in the world, especially when you're focused mainly on one type of accessibility [Mob18].

- **Accessibility.Cloud:** This one works a bit differently by pulling together accessibility information from lots of different places – OSM, specialized data providers, and even other apps that let users contribute [Acc25]. They really focus on making this data available through APIs, hoping to help build a whole ecosystem of apps that know about accessibility. The main thing to consider here is that the quality and detail of data from so many sources can vary.

- **Commercial Mapping Services:** Big names like Google Maps have also started adding basic accessibility info, like saying if a business has an accessible entrance or parking [Goo25]. These are used by a ton of people, which is great for visibility. However, the information often isn't detailed enough for serious planning (like, what are the specifics of the accessible toilet?), and for some features, it seems to rely more on what business owners say rather than community checks.

What I gathered from looking at these global efforts is that while there's a good amount of high-level or very specific (like wheelchair-only) data out there, finding detailed, locally checked information that covers a wider range of accessibility needs is still a big challenge.

2.1.2 National and Local Initiatives

Beyond the big global players, there are many projects that focus on specific countries or cities. These are often started by local governments, tourism groups, or disability organizations. For my project, it was really important to see what was already happening in Romania and specifically in Timișoara, to make sure I wasn't just redoing something that already existed and to see if there were any chances to collaborate. My research, conducted around April 2024, showed that there isn't an central hub for information like this, especially in Timisoara. There was a local project for people with visual impairment [Smu25]. This and further talks with professors from West University of Timisoara, FSP , Social Assistance, confirmed to me that there is indeed an informational gap in Timisoara regarding how accessible is Timisoara as an city and how there is little to none information about it online.

2.1.3 Data Acquisition and Quality

Any accessibility map is only as good as the data it contains. So, how that data is gathered and how accurate it is are super important. Some key things I considered were:

- **OpenStreetMap (OSM):** OSM is a huge collaborative map of the world, and it's the base for many mapping apps, including mine [Ope25a]. It has a system for tagging features, including accessibility ones. There are standard tags for things like `wheelchair`, `ramp`, `toilets:wheelchair`, `tactile_paving`, `hearing_loop`, and even specific `blind:*` tags for visual accessibility features [Ope25b]. Using OSM data can give a good starting point for finding locations, but the level of detail and how up-to-date the accessibility information is can really vary from place to place. It often needs more specific information added on top.

- **Crowdsourcing:** Enabling the user community to collect, contribute, and update data is a really powerful and scalable method, particularly effective for capturing localized and specific details that might be missed by centralized efforts. This approach is a core strategy for platforms like Wheelmap [SV12] and forms the basis of the "Timișoara Access Map" project. However, the effectiveness of crowdsourcing is based on the implementation of robust mechanisms to accept and guide users, the establishment of processes to validate submitted data, the development of strategies to resolve conflicting reports and the promotion of motivation for sustained community participation [SV12]. The administrative review process implemented in the

"Timișoara Access Map" is one key mechanism specifically designed to enhance data quality within this crowdsourced model.

- **Official and Authoritative Sources:** Sometimes you can get data from city databases, building surveys, or directly from business owners. This data can be very reliable for what it covers, but it's often hard to get hold of, to mix with other data, or to keep updated regularly.
- **Data Modeling:** It's important to have a clear and well-thought-out way to organize the accessibility information you collect. For my project, as I'll explain more in Chapter 3, I decided to use a set of simple boolean flags (yes/no questions) for key features like ramps, accessible WCs, parking, accessible entrances, Braille, audio guides, and staff help. This makes it more straightforward to collect and filter the data consistently.

2.2 Core Technologies and Frameworks

Building a web mapping application that's dynamic and interactive involves putting together several key pieces of technology for both the server-side (backend) and what the user sees in their browser (frontend).

2.2.1 Backend Development

The backend is where the main application logic lives, where data is managed, and where API requests are handled.

- **Flask Framework:** For this project, I chose Flask, which is a "microframework" for Python [Pal25a]. I liked that it's lightweight and flexible. It gives you the basics but doesn't force a lot of extra stuff on you, so you can pick and choose the tools (extensions) you need. This felt like a good fit for a project like this where I wanted control over the components. As detailed in my `app/__init__.py`, I integrated several Flask extensions like Flask-SQLAlchemy for the database work, Flask-Migrate to help with database changes, Flask-Login for user accounts, Flask-WTF for forms, Flask-Babel for languages, and Flask-Limiter for managing request rates.
- **Database and ORM:** I opted for a relational database because structuring data like user information, locations, and reviews it's simpler. To work with the database in Python, I used SQLAlchemy [SQL25], which is an Object-Relational Mapper (ORM). It lets me work with my database tables as if they were Python objects, which simplifies even further the process of writing queries and managing data.
- **API Design:** I also built a RESTful API as part of the backend (you can see the code in `app/routes/api.py`). This allows the frontend part of my application to get data dynamically and could also be used in the future if, say, someone wanted to build a mobile app that uses the same accessibility data.

2.2.2 Frontend Development

The frontend is everything the user sees and interacts with in their web browser, including the map itself.

- **Leaflet.js Library:** Leaflet is a really popular open-source JavaScript library for making interactive maps, and it's what I used for the "Timișoara Access Map" [Vla25]. It handles things like displaying the map tiles (from OSM), putting markers on the map, showing popups when you click on a marker, and responding to map events. I found it to be pretty powerful, well-documented, and relatively easy to work with, especially for creating custom-looking markers.

- **HTML/CSS/JavaScript:** These are the standard web technologies. HTML provides the basic structure of the pages (I used Jinja2 templates with Flask to generate the HTML). CSS (in `static/css/style.css` and `responsive.css`) handles all the visual styling and making sure the site fits properly on different screen sizes. And then JavaScript (my custom files like `map.js` and `app.js` in `static/js/`) makes the page interactive, handles things like user input, and communicates with the backend API.

- **Map Data Source:** As mentioned, OpenStreetMap (OSM) provides the actual base map images that Leaflet displays [Ope25a]. It's a great resource because it's a global, community-driven project, so the map data is generally quite good and up-to-date.

2.3 Accessibility Standards and Guidelines

Since this whole project is about accessibility, it was really important to me that the website itself was also accessible.

- **Web Content Accessibility Guidelines (WCAG):** These guidelines come from the W3C's Web Accessibility Initiative (WAI) and are basically the international standard for making web content accessible to people with all sorts of disabilities [W3C25b]. The main ideas are that content should be Perceivable (e.g., can you see it or have it read to you?), Operable (e.g., can you use it with a keyboard?), Understandable (e.g., is it clear and predictable?), and Robust (e.g., does it work with assistive technologies like screen readers?). I aimed to follow the WCAG 2.1 AA level criteria as much as possible because I believe it's crucial for a tool like this to be usable by everyone it's intended to help.

- **Accessible Rich Internet Applications (WAI-ARIA):** ARIA [W3C25a] is a set of special attributes you can add to HTML to make dynamic web content and custom widgets more accessible, especially for screen reader users. I used ARIA attributes for things like the interactive map elements, my custom filter buttons, and the dynamic notification messages to give them better semantic meaning.

2.4 Positioning of the Thesis Work

So, where does my "Timișoara Access Map" project fit into all this? I see this thesis as a practical contribution to the area of digital accessibility mapping. While I took some inspiration from big global platforms like Wheelmap.org and used well-established technologies like Flask and Leaflet, my focus was on addressing the specific need for detailed, locally relevant, and community-checked accessibility information right here in

Timișoara where I believe there's a real gap for this kind of information. By building in features for users to submit quite specific accessibility details, add photos, write reviews, and by having an admin step to check the data, I'm hoping the "Timișoara Access Map" can become a reliable tool that makes it easier for people to navigate our city and feel more included. I also aimed at making sure that the platform itself was built with web accessibility (WCAG) in mind.

Chapter 3

Design and Architecture

After establishing why this project is needed in today's society (Chapter 1) and looking at what already has been done (Chapter 2), this chapter will showcase how I actually planned and structured the "Timișoara Access Map" application. The overall design of the system, the thinking behind the database structure, how I designed the API for data exchange, and some of the key considerations I had to make the user interface (UI) and user experience (UX) as good as possible. My main aim here was to lay a solid foundation that would allow me to build a robust, scalable, and user-friendly platform.

3.1 System Architecture

For the "Timișoara Access Map," I decided on a fairly standard but effective multi-tier web application architecture. You can see a high-level overview of this in Figure 3.1. In summary, I separated things into a presentation tier (what the user sees and interacts with in their browser), an application tier (the backend server where all the main logic happens), and a data tier (where all the information is stored). I found this separation really helpful for keeping things organized and easier to manage as the project grew.

3.1.1 Presentation Tier (Frontend Client)

The frontend is where user's interact with the web application. It's responsible for showing the user interface and handling how they interact with the application. To build this, I used the standard trio of web technologies:

- **HTML5:** This provides the basic structure for all the web pages. Flask uses the Jinja2 templating engine, so the HTML pages are generated dynamically on the server based on these templates.
- **CSS3:** For all the visual styling – how things look, the layout, and making sure it works well on different screen sizes (responsiveness) – I wrote custom CSS files (`static/css/style.css` and `static/css/responsive.css`). This also includes the styles for the high-contrast mode and the adjustable text sizes for better accessibility.
- **JavaScript (ES6+):** This is where all the client-side interactivity comes from. It handles things like updating parts of the page without a full reload and talking to the backend API.

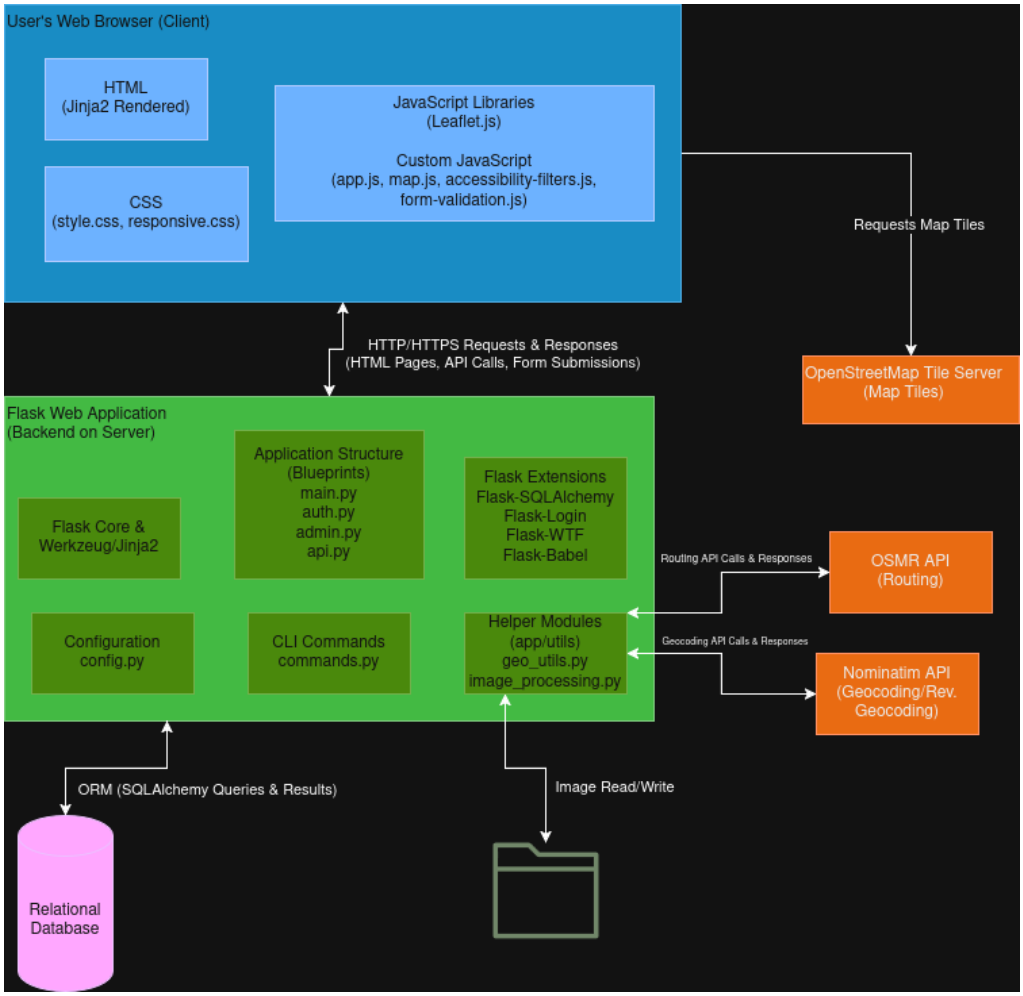

Figure 3.1: High-Level System Architecture of Timişoara Access Map.

– **Leaflet.js [Vla25]:** Is what was used to create the interactive maps, show the location markers with custom icons and popups, and handle map events like clicks and pans. The base map tiles themselves come from OpenStreetMap [Ope25a].
– **Custom Scripts (`static/js/`):** I wrote several JavaScript files to manage different parts of the frontend:
  * `app.js`: This script handles global things like the mobile navigation menu, the accessibility tools (high-contrast mode, text size), notifications, and some general keyboard interactions.
  * `localStorage` deals with saving and loading user preferences like high-contrast mode to their browser's.
  * `map.js`: This one is all about the main map. It initializes the Leaflet map, loads location data by calling the backend API, implements the search feature (again, talking to an API endpoint), and handles showing location details when a user clicks on a marker. It also manages the central marker that users can use when they're adding a new location through the main map interface.
  * `accessibility-filters.js`: was created to specifically manage how the accessibility and location type filters work. It keeps track of what

filters are active, updates the filter buttons visually, saves the user's filter choices to `localStorage`, provides ARIA announcements for screen readers when filters change, and then tells `map.js` to update the map.

* `form-validation.js`: For forms that have a 'data-validate="true"' attribute, this script provides client-side validation. This means users get instant feedback if they make a mistake in a form, and it helps make sure the data is in a good format before it even gets sent to the server. It also helps with accessibility by updating ARIA attributes for errors.

So, users see the map, use the filters, fill out forms (like for logging in, registering, or submitting a location or review), and these actions either submit data directly as an HTML form or trigger JavaScript to call the backend API for dynamic updates.

3.1.2 Application Tier (Backend Server)

The backend is where the core logic of the "Timișoara Access Map" resides. It was built using Python and the Flask microframework [Pal25a]. The backend is responsible for processing requests from the user's browser, handling all the business rules, managing user accounts and permissions, interacting with the database, and sending back data or rendered HTML pages. To keep the backend code organized, I structured it using Flask Blueprints, with the main logic located in the `app/routes/` directory:

- **`main` Blueprint (`app/routes/main.py`):** This is used in the main user-facing parts of the site, like the index page with the map, user dashboards and profiles, the forms for submitting new locations, and the pages that show detailed information about a specific location. It handles the data from submitted forms, including processing any uploaded images (like resizing them and giving them secure filenames) using a `process_and_save_image` function. These images are then stored on the server's filesystem as defined in the configuration.

- **`auth` Blueprint (`app/routes/auth.py`):** This blueprint handles everything to do with user accounts – logging in, logging out, and new user registration. I used Flask-WTF forms here for a secure way to handle this. It also manages users' language preferences.

- **`admin` Blueprint (`app/routes/admin.py`):** Was created in order to have a separate section for administrators. From here, they can moderate content (like approving or rejecting new locations, which also involves deleting the associated image files from the server if a location is rejected), manage users (like activating/deactivating accounts or changing their admin status), oversee reviews, directly delete photos, and view logs of important administrative actions (using the `AdminLog` model).

- **`api` Blueprint (`app/routes/api.py`):** This blueprint exposes a RESTful API. This is how the frontend JavaScript gets data dynamically, for example, to load locations onto the map, perform searches, or display location details. It also allows for submitting new locations and reviews programmatically.

To add these functionalities, several helpful Flask extensions we're used, which I set up in the main `app/__init__.py` file:

- **Flask-SQLAlchemy [Pal25b]:** This makes working with databases in Flask much easier by providing an Object-Relational Mapper (ORM).

- **Flask-Migrate [Gri25]:** This extension, using Alembic underneath, helps manage changes to the database structure over time. So, if I need to add a new column to a table, Flask-Migrate helps apply that change safely.

- **Flask-Login [Fra25]:** This handles all the user session management – keeping track of who is logged in and providing the `current_user` object that I use throughout the app to check permissions.

- **Flask-WTF [YJ25]:** Was used for creating and processing web forms. It also provides important security features like CSRF (Cross-Site Request Forgery) protection, which is crucial.

- **Flask-Babel [Ron]:** Because I wanted the application to be available in both Romanian and English, I used Flask-Babel for internationalization and localization. It uses a `localeselector` function to determine which language to show the user.

- **Flask-Limiter [Sai25]:** To prevent abuse and keep the application stable, I used Flask-Limiter to set rate limits on how often certain routes and API endpoints can be accessed.

Helper modules we're created in the `app/utils/` directory for specific tasks:

- **`geo_utils.py`:** This module contains functions for geospatial calculations, like figuring out the distance between two points, checking if coordinates are within Timișoara, and using the Nominatim API for geocoding (turning addresses into coordinates) and reverse geocoding. I also used `lru_cache` here to cache results from Nominatim, which helps with performance and reduces how often I hit their API. It also has code to talk to an OSRM API for route planning.

- **`image_processing.py`:** This has more advanced functions for working with images, like `save_uploaded_image` which can handle rotating images based on their EXIF data, resizing them, and compressing them. It also has functions for creating thumbnails and deleting images from the server's filesystem.

- **`validators.py`:** Here, I defined some custom validation rules that I could use with my Flask-WTF forms, for example, to check if location coordinates are valid, if an uploaded image file is of the right type and size, or if a password meets certain strength criteria.

The application's overall behavior is controlled by settings in `config.py`, which can be different for development and production environments (using the `FLASK_ENV` variable). This file uses `python-dotenv` to load sensitive information like database connection strings or the `SECRET_KEY` from a separate `.env` file. Finally, some command-line interface (CLI) commands we're written in `commands.py` (like `create-admin` and `seed-data`) to make it easier to set up and manage the application during development.

3.1.3 Data Tier

The data tier is where all the application's information is stored long-term. For this project, it's made up of two main parts:

- **Relational Database:** This is the main storage for all structured data. SQLite was used for development because it's easy to set up, but the application is configured (via `config.py`) so it could be switched to something more robust like PostgreSQL or MySQL for a production environment. This database holds all the user accounts, details about each location and its accessibility features, the metadata for photos, user reviews, and the logs of what administrators do. As mentioned, Flask-SQLAlchemy is the ORM I used to interact with this database.
- **Server Filesystem:** When users upload images for locations, those image files themselves are stored directly on the server's filesystem. The path for this is defined in the configuration as the `UPLOAD_FOLDER` (for example, a folder like `app/static/uploads/`). The database (specifically, the `Photo` model) then just stores information *about* these images, like their filenames and descriptions, and links them to the right locations.

3.1.4 External Services Integration

As shown in the system architecture diagram (Figure 3.1), the "Timișoara Access Map" also connects with a few external services to get information it needs:

- **OpenStreetMap (OSM) Tile Servers:** The Leaflet.js library on the frontend directly pulls map tiles (the actual map images) from OSM's tile servers to display the base map that users see and interact with.
- **Nominatim API (OpenStreetMap):** The backend code in `app/utils/geo_utils.py` uses the Nominatim API. This service helps turn street addresses into geographic coordinates (latitude and longitude) and can also do the reverse – turn coordinates into an address. To make this more efficient and also it's good practice to not hit their API extensively, caching was added for the results using Python's `functools.lru_cache`.
- **OSRM (Open Source Routing Machine) API:** The `app/utils/geo_utils.py` module also has the capability to connect to an OSRM API. This kind of service can calculate routes between two points, which could be really useful for planning accessible journeys. Although the "Get Directions" button on the location detail page currently just links to Google Maps for simplicity, I built in the backend logic for OSRM so it could be used for more advanced routing features in the future or for other internal calculations.

3.2 Database Structure

The way the data is organized in the database is really important for how the whole application works. I used SQLAlchemy to define the database schema in `app/models.py`. Figure 3.2 provides a visual overview of the main tables (which SQLAlchemy calls models) and how they're related to each other.

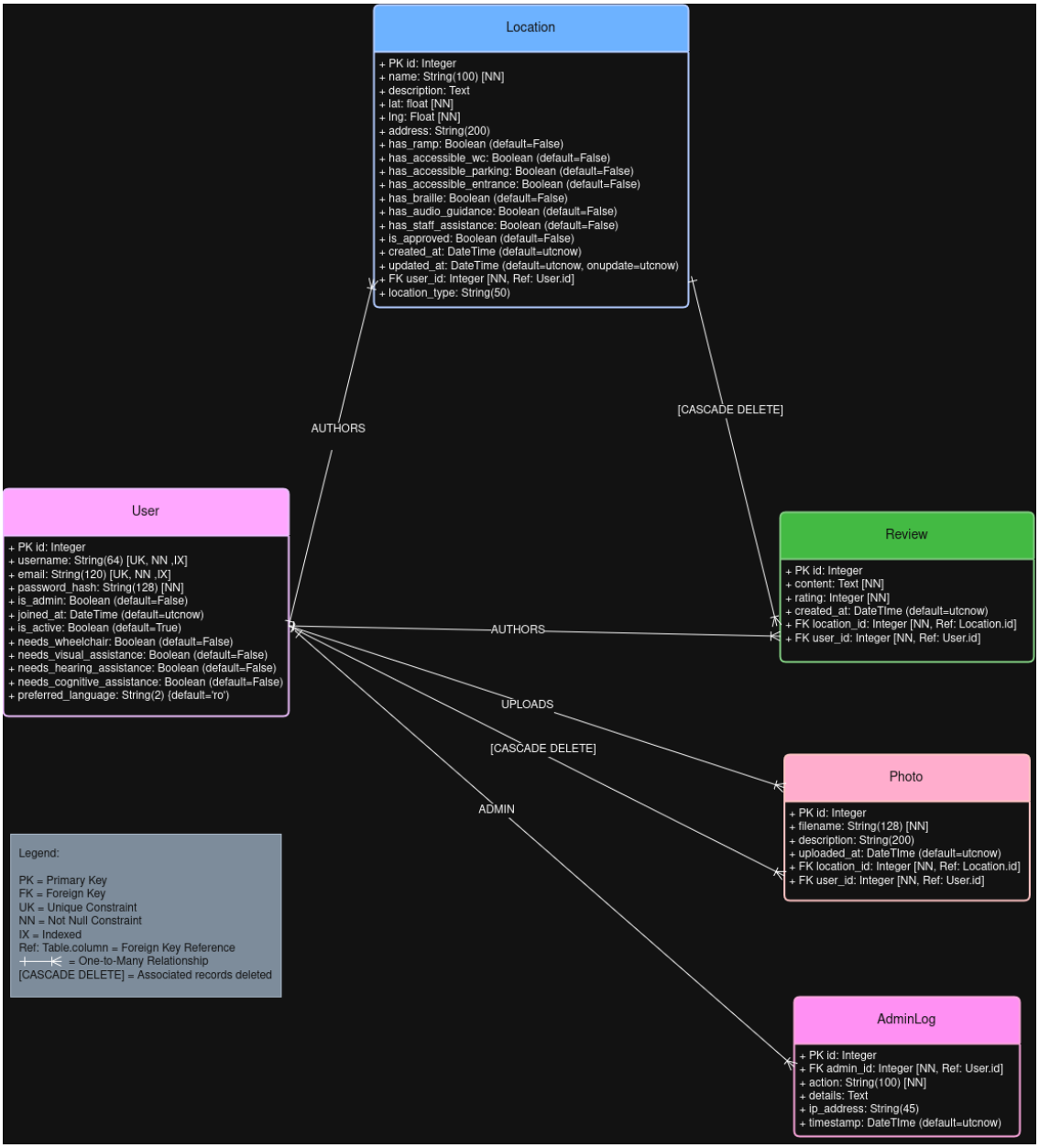

Figure 3.2: Entity-Relationship Diagram for Timişoara Access Map.

The key tables in the database are:

- **User Model (`User`):** This table stores all the information about the users registered on the platform. This includes their login details (username, email, and a securely hashed password using `werkzeug.security`), whether they are an administrator (`is_admin`), if their account is active (`is_active`), and their personal accessibility preferences (like `needs_wheelchair` or their `preferred_language`). It also keeps track of when they joined. The User model is connected to several other tables: it's linked to the locations they've submitted (as the `author`), the reviews they've written, the photos they've uploaded (through the `Photo.user_id` field), and the admin logs generated by them if they are an administrator.

- **Location Model (`Location`):** This table holds all the information about the accessible places. For each location, it stores its name, a description, its exact geographi-

cal coordinates (`lat` and `lng`), a street address, and the boolean flags (true/false values) for specific accessibility features like `has_ramp`, `has_accessible_wc`, `has_braille`, etc. It also tracks whether the location has been approved by an administrator (`is_approved`), what type of place it is (e.g., "restaurant", "shop"), when it was created and last updated, and, importantly, a foreign key linking it back to the `User` who originally submitted it.

- **Photo Model (`Photo`):** This table doesn't store the images themselves (those are on the filesystem), but it stores important metadata about each uploaded photo. This includes the `filename` (so the system can find the actual image file), an optional `description` of the photo, when it was uploaded, and foreign keys that link the photo to the `Location` it belongs to and to the `User` who uploaded it.
- **Review Model (`Review`):** This table is for user-generated reviews. Each review has its main 'content' (the text of the review), a 'rating' (from 1 to 5 stars), a timestamp for when it was created, and foreign keys linking it to the 'Location' being reviewed and the 'User' who wrote it.
- **AdminLog Model (`AdminLog`):** To keep track of important administrative actions, `AdminLog` was created this table. It logs what `action` was performed (like "`approved_location`"), any relevant `details`, the `ip_address` of the administrator at the time, a `timestamp`, and a foreign key to the `User` (who must be an admin) that performed the action.

The relationships between these tables are defined using foreign keys and SQLAlchemy's 'db.relationship' feature. For example, when I defined the relationship between a 'Location' and its 'Photo's and 'Review's, I configured it with 'cascade='all, delete-orphan''. This is a data integrity feature that means if a location is deleted from the database, all of its associated photos and reviews will automatically be deleted too. For efficiency, I also added database indexes ('index=True') to fields that are frequently searched, like the 'username' and 'email' in the 'User' table.

3.3 API Design

To make the frontend dynamic and to allow for potential future uses (like a mobile app), a RESTful API was designed and implemented. This API is built within the `api` Blueprint (`app/routes/api.py`). All the API endpoints return data in JSON format and have rate limiting applied using Flask-Limiter to prevent abuse. Here are the main endpoints:

- **`GET /api/locations`:** This is a key endpoint for fetching a list of all approved locations. It's quite flexible because it accepts query parameters to filter the results. For example, the frontend can request only locations that are wheelchair accessible ('wheelchair=true') or that are of a certain type ('type=restaurant'). This is what powers the main map display when users apply filters.
- **`GET /api/locations/<int:location_id>`:** When a user wants to see more details about a specific location (e.g., after clicking a marker on the map), the frontend calls this endpoint with the location's ID. It returns all the detailed information for that single location, including its attributes, any associated photos

(filenames and descriptions), and all user reviews. Access to locations that haven't been approved yet might be restricted through this endpoint, depending on whether the user is an admin or the original submitter.

- **`GET /api/search?q=<query>`:** This endpoint provides search functionality. The frontend sends a search term (query), and the backend searches for approved locations where the name, description, or address matches the query (using a case-insensitive partial match, or 'ilike'). It returns a list of basic details for matching locations (like ID, name, address, coordinates, and type), which the frontend then uses to show dynamic search results in the map interface. I put in a requirement for the search query to be at least a few characters long (e.g., 3 characters) to avoid overly broad searches.

- **`POST /api/submit-location` (Authenticated):** For users who are logged in, this endpoint allows them to submit information about a new location using a JSON payload in the request body. The backend takes this data, creates a new 'Location' entry in the database (which will usually be marked as unapproved initially, unless an admin submitted it), and then sends back a success message including the ID of the newly created location. This offers a programmatic way to add locations, which could be useful for other tools or a mobile app, in addition to the standard HTML form.

- **`POST /api/add-review/<int:location_id>` (Authenticated):** Similar to submitting a location, logged-in users can use this endpoint to add a review (the review text and a star rating) to an already approved location. The request also uses a JSON payload. The backend creates a new 'Review' record and links it to the correct location and user.

- **`GET /api/reports` (Authenticated):** This is an endpoint for authenticated users (adminstrators) to get some general statistics about the platform. This could include things like the total number of approved locations, how many are pending approval, the number of registered users, total reviews, and maybe even how common certain accessibility features are (like the percentage of locations that have ramp access).

3.4 User Interface (UI) and User Experience (UX) Design

When I was designing the "Timișoara Access Map," my main focus for the user interface (UI) and user experience (UX) was to make it as straightforward, efficient, and accessible as possible for everyone, especially for people with disabilities who are the main audience for this tool.

3.4.1 Core UI Components and Workflow

I tried to organize the application into logical sections and make the common tasks easy to perform:

- **Main Map Interface (`templates/index.html`):** This is the heart of the application.

  - It's dominated by a large, interactive Leaflet map where location markers are displayed.
  - On one side, there's a collapsible sidebar (`.controls-sidebar`) that holds the search box, the main accessibility filter buttons (for wheelchair access, visual needs, etc.), checkboxes for filtering by location type (like restaurant, shop), and a map legend. If a user is logged in, this sidebar also has a button to start adding a new location.
  - When a user clicks on a map marker or a search result, another sidebar (`.details-sidebar`) slides into view from the other side, showing detailed information about that specific location. This content is loaded dynamically by calling the API.
  - The "Add Location" form itself is also embedded within a sidebar (`.form-sidebar`) on this main page. When a user wants to add a location using the button in the controls sidebar, this form slides out. The coordinates for the new location are automatically filled in based on where the center of the main map is currently positioned.

- **Location Detail Page (`templates/location/details.html`):** If a user wants to see *all* the information about a location, they can go to its dedicated detail page.
  - This page shows everything: name, address, a full description, a clear list of all its accessibility features (with "Yes/No" indicators and icons), a grid of any uploaded photos, and all the user reviews for that place.
  - It also has a smaller, embedded Leaflet map that's zoomed in on just that specific location. I made it so that scroll-wheel zoom on this small map is only enabled if the user actually clicks on or focuses on the map, to prevent accidental zooming while scrolling the page.
  - If the user is logged in and the location is approved, there's a form here for them to submit their own review, including an interactive star rating input.
  - There are also action buttons, like a "Get Directions" button (which currently links to Google Maps for simplicity) and, depending on the user's status (if they are an admin or the original submitter), there might be buttons for administrative actions like approving or editing the location.

- **Location Submission Form (`templates/location/submit.html`):** Besides adding a location from the main map interface, users can also go to a separate, dedicated page to submit a new location.
  - This page has a more detailed, multi-part form, broken down into fieldsets for location details, accessibility features, and photos.
  - It includes its own interactive Leaflet map where users can click to pinpoint the location or drag a marker, and this updates the latitude and longitude fields in the form. There's also a "Use My Current Location" button that uses the browser's geolocation feature, after checking if the user is actually within Timișoara.
  - Users can check off all the relevant accessibility features and upload multiple photos along with descriptions for them.

- **User Authentication Pages (`templates/auth/login.html`, `templates/auth/register.html`):**

  - I aimed for clean and simple forms for users to sign in and create new accounts. These are built using Flask-WTF on the backend for generating the forms and doing server-side validation, and I also used my `form-validation.js` script for immediate client-side checks on forms that have the 'data-validate="true"' attribute.
  - The registration form has a section where new users can, if they want, indicate their primary accessibility needs when they sign up. This information can then be used to, for example, pre-select some filters for them when they first use the map.

- **User Dashboard and Profile (`templates/user/dashboard.html`, `templates/user/profile.html`):**
  - The dashboard gives logged-in users a quick overview of their own activity on the site, like the locations they've submitted (and whether they're approved yet) and the reviews they've written, usually shown in a card-style layout.
  - The profile page is where users can update their account details like email and preferred language, manage the accessibility preferences they declared, and change their password. These are handled by separate forms on the page.

- **Administrative Interface (templates in the `templates/admin/` directory):**
  - I built a separate section just for administrators. It starts with a dashboard that shows key platform statistics (like how many locations are pending approval, total number of users, etc.) in easy-to-read summary cards.
  - There are pages with tables for managing all the locations, users, and reviews, with options to filter and search the data. Some of the filters in the admin area have client-side JavaScript to make them a bit more user-friendly, like automatically submitting the filter form when a dropdown selection changes.
  - A key part is the workflow for reviewing and approving (or rejecting) new location submissions (on `pending_locations.html`). Each pending location is often shown with a small, non-interactive map to give the admin quick context, along with forms for the moderation actions.
  - Admins also have forms for editing existing location details (on `edit_location.html`), which includes an interactive map for adjusting coordinates if needed, and for managing user statuses (like making someone an admin or deactivating an account).
  - There's also a page for browsing the administrative action logs (`logs.html`), again with options to filter the logs.
  - I used modals (pop-up dialog boxes) for important confirmations, like before an admin deletes a location.

3.4.2 UX Principles and Considerations

While building these components, I tried to keep a few user experience principles in mind:

- **Task-Oriented Design:** I wanted the interfaces to be structured around what users need to do: find accessible places, submit new information, manage their contributions, or, for admins, oversee the platform.

- **Progressive Disclosure:** Instead of overwhelming users with too much information at once, I often used sidebars that slide into view (for location details or the quick add form) or dedicated pages for more complex tasks. This helps keep the main map interface cleaner until a specific interaction requires more detail.

- **Responsive Design:** It was really important for the application to work well on different devices. So, the CSS (`responsive.css`) uses media queries to adapt the layout for desktops, tablets, and mobile phones. This includes things like changing the main navigation to a ”hamburger” menu on small screens and adjusting how sidebars and grids are displayed.

- **Visual Feedback:** Providing feedback for user actions is key for a good experience.
  - Filter buttons and other selectable items have clear ”active” states.
  - When an action is completed on the server (like successfully logging in or submitting a form), a ”flash message” appears at the top of the page (`base.html`).
  - For things happening on the client-side (like an error trying to load map data or an API call failing), I built a notification system (`app.js` and `map.js` use the `#notification-container`) to show dynamic alerts.
  - There are also subtle visual cues, like the pulsing selection marker on the map that shows where a new location might be added.

- **Data Persistence (for user convenience):** To make things easier for returning users, some of their settings are saved in their browser's `localStorage`. This includes their chosen map filters (`accessibility-filters.js`) and their settings for accessibility tools like high-contrast mode or text size (`app.js`).

- **Internationalization (i18n):** The user interface is designed to be translatable and currently supports English and Romanian. This is handled by Flask-Babel, and all the text that users see is wrapped in a special function (`_()` or gettext) in both the Python code and the Jinja2 templates.

- **Client-Side Validation (`form-validation.js`):** For many of the forms (like login, registration, password change, and the admin edit location form), I added client-side validation using my `form-validation.js` script. This script activates on forms that have a ‘data-validate=”true”‘ attribute. It checks the input as the user types or when they try to submit, providing instant feedback if there are errors. This improves the quality of data sent to the server and makes the form-filling process smoother for users. Some other forms, like the main location submission page (`location/submit.html`), also have simpler, inline JavaScript validation for quick checks.

3.4.3 Accessibility of the ”Timişoara Access Map” Platform

A core goal for me was to make sure that the ”Timișoara Access Map” platform itself is as accessible as possible, especially since it's a tool designed to help people with disabilities. I tried to follow the principles of the Web Content Accessibility Guidelines (WCAG) 2.1 [W3C25b].

- **Semantic HTML:** I focused on using HTML5 elements correctly to give the content a clear structure that assistive technologies can understand (e.g., using `<header>`, `<nav>`, `<main>`, `<aside>`, `<button>`, `<label>`, and grouping related form fields with `<fieldset>` and `<legend>`). The language of the page (`lang` attribute on the `<html>` tag) is also set dynamically based on the user's choice.

- **Keyboard Navigability:** I made sure that all interactive parts of the website—links, buttons, form fields, map controls, and my custom filter buttons—can be reached and used with just a keyboard. There's also a "skip to content" link (`.skip-link`) at the top of every page so keyboard users can quickly get to the main part of the page. My `app.js` script also helps make keyboard focus more visible by adding a `.keyboard-navigation` class to the body when the 'Tab' key is used.

- **ARIA (Accessible Rich Internet Applications) Roles and Attributes [W3C25a]:** For parts of the site that are dynamic or have custom interactions, I used ARIA attributes to provide more information to assistive technologies:
  - `aria-expanded` is used for things like the mobile navigation menu to indicate if it's open or closed.
  - `aria-pressed` is used on toggle buttons, like the accessibility filters, to show their current state.
  - `aria-controls` helps link a control (like a button) to the content it manages.
  - `aria-label` is used for buttons that only have icons (like some search or admin action buttons) or for links where the text might not be descriptive enough on its own, to give them an accessible name.
  - `aria-live="polite"` is used for areas of the page that update dynamically, like the notification container (`#notification-container`) or the dynamically created status announcers for filters (`#filter-status-announcer`) and general accessibility changes (`#accessibility-announcer`). This tells screen readers to announce the changes without interrupting what the user is currently doing.
  - `role="alert"` is used for important messages, like the server-sent flash messages, to make sure they are announced.
  - For form validation errors, `aria-invalid` is set on the input field, and `aria-describedby` is used to programmatically link the input to its specific error message. This is handled by my `form-validation.js` script.
  - The main map container (`#map`) itself has an `aria-label` to describe its purpose.

- **Visual Accessibility Tools (handled by `app.js` and CSS):**
  - **High-Contrast Mode:** Users can toggle a high-contrast theme for the website. This applies a `.high-contrast` class to the body, which then uses a different set of color variables defined in `style.css` to improve readability for users with low vision.
  - **Adjustable Text Size:** There are buttons that let users increase or decrease the base font size of the application. This works by toggling `.large-text` and `.xl-text` classes on the body, which then affects font sizes that are defined using CSS variables.

- **Sufficient Color Contrast:** In both the default and high-contrast themes, I aimed to ensure that the contrast between text and its background meets WCAG AA success criteria, making text easier to read.
- **Reduced Motion (handled by `responsive.css`):** For users who are sensitive to motion, the CSS includes an `@media (prefers-reduced-motion: reduce)` query. This rule minimizes or disables non-essential CSS animations and transitions.
- **Image Accessibility:** When users upload photos for locations, they can add descriptions, and these descriptions are used as alternative text for the images. Decorative icons on the site are generally part of labeled controls or use an `aria-label` if they are standalone interactive elements. I also implemented lazy loading (`loading="lazy"`) for images on the location details page to help with page performance.
- **Form Accessibility:** All form inputs are properly associated with `<label>` elements. Related groups of controls are often grouped using `<fieldset>` and `<legend>`. As mentioned, validation errors are programmatically linked to their inputs.

My goal with all these design choices was to make the ”Timișoara Access Map” a tool that is not only useful in its content but also usable and welcoming for the widest possible range of people, especially those with disabilities. The idea is that a platform *about* accessibility should really practice what it preaches.

Chapter 4

Implementation

This chapter details the technical realization of the "Timișoara Access Map," translating the architectural design outlined in Chapter 3 into a functional web application. It covers the development environment setup, backend implementation using Flask and its extensions, frontend development with HTML, CSS, and JavaScript (including Leaflet.js for map functionalities), and the database integration.

4.1 Development Environment and Tooling

The project was developed using Python 3.10.12 . A key aspect of the development setup was the use of a Python virtual environment (`venv`), created using Python's native `venv` module. This ensured that all project dependencies were isolated from the system's global Python installation, promoting reproducibility and avoiding version conflicts. All required external libraries were managed via `pip` and listed in a `requirements.txt` file.

Core development tools included:

- **Code Editor/IDE:** Visual Studio Code 1.67.2.
- **Browser Developer Tools:** Firefox Developer Tools (Version 136.0.4) for frontend debugging and inspection.
- **Flask Version:** 2.2.3

4.2 Backend Implementation

The backend logic is implemented using the Flask microframework in Python, following the factory pattern (`create_app` function in `app/__init__.py`) for better organization and testability.

4.2.1 Flask Application Setup and Core Structure

The main application instance is created within the `create_app` function located in `app/__init__.py`. This function is responsible for:

1. Initializing the Flask application (`app = Flask(__name__)`).

2. Loading configuration settings from a `config.py` file, which itself uses `python-dotenv` to load environment variables for sensitive data like `SECRET_KEY` and `DATABASE_URL`.

3. **Initializing Flask Extensions:** Each Flask extension is instantiated globally and then initialized within `create_app`:

    - **Flask-SQLAlchemy (`db`):** For ORM capabilities, connecting to the database specified in the configuration.
    - **Flask-Migrate (`migrate`):** Integrated with Flask-SQLAlchemy and Alembic to manage database schema migrations. Commands like `flask db init`, `flask db migrate`, and `flask db upgrade` were utilized during development.
    - **Flask-Login (`login_manager`):** Configured for user session management. This includes setting the login view to `'auth.login'`, defining login messages, and registering the `@login_manager.user_loader` callback in `app/models.py`.
    - **Flask-WTF (`csrf`):** Provides global CSRF protection and facilitates form creation and validation.
    - **Flask-Limiter (`limiter`):** Configured with `get_remote_address` to apply rate limits to routes and API endpoints, protecting against abuse.
    - **Flask-Babel (`babel`):** Initialized for internationalization, with a `localeselector` function (`get_locale` in `app/__init__.py`) supporting 'ro' and 'en' based on session and browser preferences.

4. **Registering Blueprints:** Routes are modularized using Flask Blueprints (`main.py`, `auth.py`, `admin.py`, `api.py` from `app/routes/`), registered with URL prefixes where appropriate (e.g., `/auth`, `/admin`, `/api`).

5. **Database Initialization:** `db.create_all()` is called within the application context for initial table creation during early development; subsequently, Flask-Migrate handles schema management.

6. **Context Processors:** A context processor (`inject_global_variables`) makes `current_year` available to all Jinja2 templates.

7. **Registering CLI Commands:** Custom commands (`create-admin`, `seed-data` from `commands.py`) are registered for setup and management via the `flask` tool.

4.2.2 Database Models (`app/models.py`)

The data persistence layer relies on SQLAlchemy models defined in `app/models.py`, reflecting the ERD in Figure 3.2.

- The `User` model utilizes `werkzeug.security` for password hashing (`set_password`, `check_password`) and inherits `UserMixin` for Flask-Login integration.

- The `Location` model structures accessibility features as boolean flags and includes a `to_dict()` method for JSON serialization for API responses.
- Relationships (e.g., `User.locations`, `Location.photos`) are defined using `db.relationship` with `backref` attributes and `lazy='dynamic'` loading. Critical for data integrity, 'cascade='all, delete-orphan'' is set for `Location`'s dependent `Photo` and `Review` records.

4.2.3 Routing and View Functions (`app/routes/`)

Each blueprint contains specific routes and view functions:

- **Main Blueprint (`main.py`):** Implements core user-facing logic.
  - The `/submit-location` route handles both GET requests (rendering the form) and POST requests (processing submissions). Image uploads within this route utilize `werkzeug.utils.secure_filename` and a local `process_and_save_image` function for resizing and saving. Non-admin submissions are set to `is_approved=False`.
  - The `/location/<int:location_id>` route dynamically renders detailed location pages, with access controls for unapproved content.
  - The `/profile` route displays the authenticated user's profile page and handles POST requests for updating basic account settings like email and preferred language. Dedicated POST routes like `/update-preferences` and `/change-password` allow users to manage their specific accessibility needs (which can influence filtering) and update their security credentials, enabling user personalization and account management.
- **Auth Blueprint (`auth.py`):** Manages user authentication.
  - Leverages Flask-WTF forms (`LoginForm`, `RegistrationForm`) with custom field validators.
  - Implements login, logout, registration, and language switching functionalities.
- **Admin Blueprint (`admin.py`):** Provides administrative tools.
  - Routes are protected by the `@admin_required` decorator.
  - Admin actions are logged via a `log_admin_action` helper to the `AdminLog` model.
  - Enables moderation of locations (approve/reject, including filesystem photo deletion on reject), user management, and review/photo deletion.
- **API Blueprint (`api.py`):** Exposes RESTful endpoints.
  - Endpoints like `/api/locations` support filtering using SQLAlchemy's `or_`, `and_` constructs. Search (`/api/search`) uses `ilike`.
  - Authenticated endpoints allow programmatic submission of locations and reviews.

4.2.4 Utility Modules (`app/utils/`)

Reusable functionalities are encapsulated in helper modules:

- **`geo_utils.py`:** Contains geospatial functions like `calculate_distance`, `is_within_timisoara`, and API calls to Nominatim (with `functools.lru_cache`) and OSRM using the `requests` library.
- **`image_processing.py`:** The `save_uploaded_image` function uses Pillow for comprehensive image processing (EXIF rotation, resizing, compression, UUID naming). Other utilities include thumbnail creation and deletion.
- **`validators.py`:** Provides custom WTForms validators (e.g., `validate_location_coordinates`, `validate_image_file`, `validate_password_strength`).

4.3 Frontend Implementation

The frontend presents data and captures user input using HTML5, CSS3, and JavaScript.

4.3.1 HTML Templating (Jinja2)

Jinja2 is used for dynamic HTML generation.

- A common `templates/base.html` provides site structure, navigation (dynamically rendered based on `current_user` status), and global asset includes (Leaflet CDN, local CSS/JS). It uses Jinja2 blocks for content injection.
- Flash messages and global variables (via context processors) are available in all templates. Internationalization is achieved with `_()` (gettext).
- Forms utilize Flask-WTF rendering (e.g., {{ `form.username.label` }}) including {{ `form.hidden_tag()` }} or {{ `csrf_token()` }} for CSRF protection.

4.3.2 CSS Styling

Styling is managed by `static/css/style.css` (main styles, CSS custom properties, theming, high-contrast mode) and `static/css/responsive.css` (media queries for various screen sizes, reduced motion). Accessibility features like adjustable text size classes and clear focus styles are defined.

4.3.3 JavaScript Implementation (`static/js/`)

Client-side interactivity is driven by several JavaScript modules:

- **`app.js` (`AccessibilityApp` class):** Manages global UI (mobile menu, accessibility tools like high-contrast/text-size toggles with ‘localStorage‘ persistence and ARIA announcements), notifications, and global keyboard listeners (e.g., ’Escape’ key).

- **`map.js` (`AccessibilityMap` class):** Initializes and controls the main Leaflet map ('window.map'). Handles loading locations and search via API calls, dynamic display of details in a sidebar, custom marker creation, management of the central selection marker, and applying filters by calling `loadLocations()`. Implements `getUserLocation()` via browser geolocation.

- **`accessibility-filters.js` (`AccessibilityFilters` class):** Manages filter button/checkbox states, persists choices in 'localStorage', provides ARIA announcements, and calls `window.map.applyFilters()` to update the map. Can be initialized with user profile preferences.

- **`form-validation.js` (`FormValidator` class):** Provides client-side validation for forms marked with `data-validate="true"`, displaying inline errors and updating ARIA attributes. Contains rules for various input types including custom coordinate checks.

- **Inline Scripts in Templates:** Page-specific JavaScript is used for tasks like initializing smaller maps (e.g., in admin views, location submission page), modal dialog logic, or unique UI interactions on specific pages (e.g., the "Add Location" button on `index.html` interacting with the main map and its embedded form).

4.4 Key Implementation Challenges and Solutions

4.4.1 Challenge 1: Ensuring Consistent Display of User-Uploaded Images with Various Orientations

Problem I Faced: When users upload photos for locations, particularly those taken with smartphones or digital cameras, these images often include EXIF (Exchangeable image file format) metadata. A common piece of this metadata is an 'Orientation' tag, which indicates if the image should be rotated or flipped to be viewed correctly (for example, a photo taken in portrait mode might be stored as a landscape image with an EXIF tag specifying a 90-degree rotation). I discovered that web browsers are inconsistent in how, or even if, they interpret and apply this EXIF orientation data when rendering an image in an `<img>` tag. This inconsistency led to a significant user experience problem: some uploaded photos were appearing sideways or even upside down on the website. This not only looked unprofessional but also diminished the usefulness of the visual information users were trying to share.

Solution Adopted: To ensure all uploaded images display correctly regardless of their original EXIF orientation, I implemented a server-side image processing step that automatically re-orients images before they are saved. This logic was primarily encapsulated within my $\texttt{app/utils/image}_{p}rocessing.pymodule, utilizingthePillow(PIL)libraryinPython.$

The core of the solution involved these steps within the $\texttt{save}_{p}rocessed_{i}magefunction, whichcallsadedicatedhelper,$

```
**Value 1 (Normal):** No transformation needed.

**Value 2 (Flipped Horizontally):** Applied image_-
obj.transpose(Image.FLIP_LEFT_RIGHT).
```

```
**Value 3 (Rotated 180°):** Applied image_obj.rotate(180).

**Value 4 (Flipped Vertically / Rotated 180°
+ Flipped Horizontally):** Applied image_-
obj.rotate(180).transpose(Image.FLIP_LEFT_RIGHT).

**Value 5 (Transposed - Flipped Horizontally +
Rotated 90° CW):** Applied image_obj.rotate(90,
expand=True).transpose(Image.FLIP_LEFT_RIGHT).

**Value 6 (Rotated 90° CCW / 270° CW):** Applied image_-
obj.rotate(90, expand=True).  (Pillow's 'rotate' is
counter-clockwise by default, so 'rotate(90)' effectively
handles the EXIF tag for 270° clockwise rotation relative
to visual top).

**Value 7 (Transposed - Flipped Horizontally + Rotated
270° CW / 90° CCW):** Applied image_obj.rotate(-90,
expand=True).transpose(Image.FLIP_LEFT_RIGHT).

**Value 8 (Rotated 90° CW):** Applied image_obj.rotate(-90,
expand=True).
```

For rotations of 90° or 270° (-90°), I used the `expand=True` argument with `img.rotate()`. This was crucial because these rotations swap the image's width and height. Without 'expand=True', Pillow would try to fit the rotated image into the original dimensions, potentially cropping it.

Robust Error Handling: The `auto_rotate_image` function includes a `try-except` block to catch potential errors during EXIF processing (e.g., `AttributeError` if `_getexif` returns 'None', or `KeyError`/`IndexError` if the 'Orientation' tag is missing or malformed). In such cases, a warning is logged, and the original, unrotated image is used, preventing the entire upload from failing due to problematic EXIF data.

Integration into Save Process: This `auto_rotate_image` function is called within the main `save_processed_image` function immediately after the image is opened from the uploaded file stream and *before* any resizing or other processing steps.

Saving the Corrected Image: After the potential auto-rotation (and subsequent steps like alpha channel conversion and resizing, also handled in `save_processed_image`), the modified Pillow `Image` object, now in its visually correct orientation, is saved to the server's filesystem with a unique filename.

4.4.2 Challenge 2: Implementing Dynamic and Accessible User Feedback for Form Submissions and Client-Side Actions

Problem I Faced: The application has many forms (login, registration, location submission, reviews) and client-side interactions (like applying filters). I needed a way to give users clear and immediate feedback about what was happening – whether an action was successful, if there was an error, or if something was loading. Just relying on page reloads for feedback can feel slow. Furthermore, this feedback needed to be accessible, especially for users relying on screen readers who might not see visual changes on the page.

Solution Adopted: I implemented two main types of feedback mechanisms:

1. **Server-Side Flash Messages:** For actions that involve a full form submission to the server (like logging in, registering, or submitting a location via a standard HTML POST), I used Flask's flash() messaging system. This allows the backend to send a short message (e.g., "Login successful!", "Location submitted for review.") that is then displayed at the top of the next page the user sees (rendered in `base.html`). I styled these messages with different colors for success, error, or information and made them closable. For accessibility, the container for these flash messages has an ARIA role="alert" to ensure screen readers announce them.

2. **Client-Side Notifications and ARIA Live Regions:** For actions happening primarily on the client-side with JavaScript (like applying map filters or an error occurring during an API call made via fetch), a page reload isn't desirable. For these, I:

   - Developed a `showNotification` function in `app.js` that dynamically creates and displays temporary notification messages (often in a corner of the screen) without a full page refresh. These also have an ARIA role="alert".
   - For filter changes and accessibility tool toggles (like high contrast), I used JavaScript to update dedicated ARIA live regions (role="status" and aria-live="polite"). These are hidden div elements whose content changes are automatically announced by screen readers, providing crucial non-visual feedback (e.g., "Wheelchair filter enabled," "High contrast mode disabled"). This was primarily managed within `app.js` and `accessibility-filters.js`.

This combination of server-sent flash messages for page-level feedback and client-side dynamic notifications/ARIA announcements for more granular interactions helped create a more responsive and accessible user experience, keeping users informed about the application's status.

4.4.3 Challenge 3: Managing Configuration for Different Environments (Development vs. Production)

Problem I Faced: As I was developing the application, I needed different settings for my local development environment compared to what would be needed for a real, live "production" deployment. For example, in development, I used a simple SQLite database and had debugging features turned on. For production, I'd want to use a more robust database (like PostgreSQL), have debugging turned off, and use a strong, secret SECRET$_KEYthatwasn'thardcodedintomyversion-controlledfiles.Managingthesedifferentconfigurationswithoutaccidentallycommittingsensitivei$

Solution Adopted: I addressed this by setting up a flexible configuration system using a combination of a config.py file and environment variables:

1. **Base Configuration Class (config.py):** I created a base Config class in `config.py` that holds all the common configuration settings (like UPLOAD$_FOLDER, ALLOWED_EXTENSIONS).Italsodefineddefaultvaluesforthingstha$
$DevelopmentConfig, TestingConfig, andProductionConfig.Eachofthesesubclassescouldov$
$TrueandmightuseanSQLitedatabasepath, whileProductionConfigwouldsetDEBUG =$
$FalseandcouldbeconfiguredtogetitsdatabaseURLandSECRET_KEYfromenvironmentvaria$

2. **Loading Configuration in app/**$_{init.py:Inmyapplicationfactory(create_appfunctionin}$ This setup provided a clean way to manage different settings for different situations. It kept sensitive

information out of my codebase, made it easy to switch between development and (theoretical) production settings, and ensured that the correct configuration was loaded based on the environment.

—

Chapter 5

Testing and Evaluation

After the main implementation phase of the ”Timișoara Access Map” (detailed in Chapter 4) was complete, a crucial next step was to thoroughly test and evaluate the application. The goal of this phase was twofold: first, to verify that all the designed features were functioning correctly and that the system was technically sound; and second, to gather some initial feedback on its usability and overall effectiveness in addressing the problem statement. This chapter outlines the testing methodologies I employed, the environment in which testing was conducted, and a discussion of the key results and findings.

5.1 Testing Methodologies

Given the scope of this thesis project and the resources available, I adopted a multi-faceted approach to testing, primarily focusing on comprehensive functional testing, supplemented by considerations for usability and accessibility.

5.1.1 Functional Testing

The main goal of functional testing was to ensure that all core features of the ”Timișoara Access Map” worked as specified in the objectives (Section 1.4) and design documents (Chapter 3). This was largely conducted through **manual end-to-end testing**, where I simulated various user scenarios and interacted with the application as a typical user or administrator would.

I developed a checklist of key functionalities and user workflows to guide this process. Some of the major areas I focused on included:

- **User Authentication and Management:** Verified registration, login/logout, profile updates, and password changes.
- **Location Submission and Management (User Perspective):** Tested submission of new locations with all features, including photo uploads and coordinate selection via map and geolocation.
- **Map Interaction and Filtering:** Ensured functionality of all filters (accessibility needs, location types), search, popups, and dynamic loading of details in the sidebar. Filter persistence via ‘localStorage‘ was also checked.
- **Reviews and Ratings:** Tested submission and display of reviews and ratings by authenticated users for approved locations.

- **Administrative Functions:** Validated admin login, dashboard statistics, approval/rejection workflows for locations (including filesystem photo handling), user management, and deletion of reviews/photos. Admin action logging was verified.

• 5.1.2 Accessibility Testing

Evaluating the platform's own accessibility involved:

– 5.2 Test Environment

Testing was primarily conducted in a local development environment:

* **Operating System:** [Pop! OS 22.04]
* **Python Version:** [3.10.12]
* **Flask Version:** [2.2.3]
* **Database:** SQLite 3.

* 5.2.1 Functional Testing Results

The manual end-to-end functional testing confirmed that the vast majority of core functionalities were operational as designed.

· User account workflows, location submission with various data types (including images), map filtering, searching, and review functionalities performed correctly.

· Administrative tasks, including content moderation and user management, were successfully validated.

· During testing, a few minor bugs were identified. For instance, when toggling high-contrast mode firstly it worked as intented but when the user went to other pages the text would match the background and became unreadable . Also one more bug was the marker not appearing onto the map, the location itself exists marked on the map but the icon isn't rendered. Due to time constraints these bugs remained unfixed.

Overall, the application demonstrated functional robustness for its intended operations.

5.2.2 Accessibility Testing Results

Accessibility testing revealed a generally good level of conformance, with some areas noted for potential enhancement:

· Keyboard Navigation: All interactive elements were confirmed to be keyboard accessible, with a logical focus order and clearly visible focus indicators. The "skip to content" link was functional.

· Visual Accessibility Tools: The high-contrast mode significantly improved text legibility with minor issues, and the text resizing controls functioned correctly, adjusting font sizes across the application.

5.2.3 Performance (Informal Observations)

No formal load or stress testing was conducted. However, informal observations during development and functional testing indicated:

- Page load times for typical views were acceptable on a standard broadband connection.
- API response times for fetching locations and search results were responsive with the development dataset size.
- The client-side map rendering with Leaflet.js handled the seeded data and several dozen additional test locations without noticeable lag.

5.2.4 Summary of Testing and Evaluation

The testing and evaluation phase demonstrated that the ”Timișoara Access Map” is a functionally sound application that fulfills its core design objectives. It provides a usable interface for users to find, submit, and review accessible locations, and for administrators to manage the platform’s content. Identified bugs were minor and will be addressed. The application, in its current state, provides a reliable foundation for a community-driven accessibility resource.

—

Chapter 6

Conclusions and Future Work

This thesis detailed the journey of designing, developing, and evaluating the ”Timișoara Access Map,” a web application aimed at addressing the significant lack of centralized, detailed, and community-driven accessibility information for the city of Timișoara. This final chapter brings the project to a close by summarizing its main achievements in relation to the objectives I set out, offering some personal reflections on the development process and acknowledging the inherent limitations of the current system, and finally, proposing potential avenues for future work and enhancements that could further build upon this foundation.

6.1 Summary of Achievements and Contribution

Looking back at the objectives I outlined in Chapter 1, I believe the ”Timișoara Access Map” project has successfully met most of its primary goals. The main contributions of this thesis and the developed application are:

- **A Functional Web Platform:** I successfully designed and implemented a fully functional web application using Python, Flask, SQLAlchemy, Leaflet.js, and standard frontend technologies. The platform allows users to register, log in, submit information about accessible locations, view these locations on an interactive map, filter them based on specific accessibility needs, and contribute reviews.
- **Crowdsourced Data Model:** The application establishes a viable model for crowdsourcing detailed accessibility information. It provides the necessary tools for users to contribute data about features like ramps, accessible restrooms, parking, entrances, Braille signage, audio guidance, and staff assistance.
- **Interactive Map with Filtering:** A core achievement is the interactive map interface that not only visualizes geolocated data but also empowers users to filter locations based on multiple accessibility criteria and location types, tailoring the information to their specific requirements.

- **Administrative Moderation System:** To help ensure data quality, a complete administrative backend was implemented. This allows for the review and approval (or rejection) of user-submitted locations, management of user accounts, and oversight of reviews and photos. This moderation workflow is crucial for maintaining the reliability of a crowdsourced platform.
- **Focus on Platform Accessibility:** Throughout the development, I made a conscious effort to adhere to web accessibility principles (WCAG, ARIA). Features like keyboard navigability, ARIA enhancements for dynamic content, user-toggleable high-contrast mode, and adjustable text sizes were implemented to make the platform itself more usable by individuals with disabilities.
- **Addressing a Local Need:** The project directly addresses a specific, identified gap in Timișoara by providing a dedicated resource for accessibility information, which was previously lacking. My hope is that this tool will genuinely enhance urban navigation and promote a more inclusive environment for residents and visitors.

6.2 Reflection and Limitations

The process of developing the ”Timișoara Access Map” from concept to a functional application has been an valuable learning experience. It provided deep insights into full-stack web development, database design, API implementation, frontend interactivity, and the practical challenges of building a user-centric application.
Some key reflections and lessons learned include:

- **The Importance of Iterative Design:** While I started with a clear set of objectives, some aspects of the UI and feature set evolved during implementation based on what felt most intuitive or what proved technically more feasible within the project's timeframe. For instance, the specific layout of the location submission form went through a couple of revisions to improve clarity and reduce cognitive load for users entering many details.
- Despite the progress made, the current implementation of ”Timișoara Access Map” has certain limitations, which are natural for a thesis project of this scope:
- **Initial Dataset Size:** The application's utility is directly proportional to the amount of accessibility data it contains. At launch, and with only the seeded demonstration data, its immediate practical value is limited. Its success hinges on active community participation to populate and maintain the dataset.
- **Moderation Scalability:** The current administrative moderation process relies on a potentially small number of administrators. As the number of submissions grows, this could become a bottleneck.

6.3 Future Work and Potential Enhancements

The ”Timișoara Access Map” platform, in its current state, provides a robust foundation that I believe can be significantly built upon. Based on my experience during its development and the limitations identified, several avenues for future work and enhancement appear promising:

- **Integration with Official Timișoara Municipal Platforms:** A highly impactful future direction would be to engage with Timișoara's municipal authorities to explore the possibility of integrating the ”Timișoara Access Map” with official city websites or information portals. Such an integration could significantly broaden the platform's audience reach, lending it an official endorsement and making accessibility information more readily discoverable byThis residents and visitors alike. Collaboration with the city administration could also provide opportunities for incorporating authoritative accessibility data and potentially securing resources for the platform's long-term maintenance and development, thereby enabling ”Timișoara Access Map” to transition from a thesis project into a sustained and officially supported public service resource for individuals with disabilities.
.
.
.

Bibliography

[Acc25] Accessibility.Cloud Contributors. Accessibility.cloud - the worldwide exchange for accessibility data. https://www.accessibility.cloud/, 2025. Accessed: 2025-06-13.

[BA19] M. Foody G. Mooney P. Basiri A., Haklay. Crowdsourced geospatial data quality: challenges and future directions. 2019. International Journal of Geographical Information Science, 33(8), 1588–1593. Accessed: 2025-05-24.

[Fra25] Matthew Frazier. Flask-login documentation. https://flask-login.readthedocs.io/, 2025. Accessed: 2025-06-10.

[Goo25] Google. Find wheelchair accessible places with google maps, 2025. Google Maps Help. Accessed: 2025-04-10.

[Gri25] Miguel Grinberg. Flask-migrate documentation. https://flask-migrate.readthedocs.io/, 2025. Accessed: 2025-06-10.

[KJH24] Efthimis Kapsalis, Nils Jaeger, and Jonathan Hale. Disabled-by-design: effects of inaccessible urban public spaces on users of mobility assistive devices – a systematic review. *Disability and Rehabilitation: Assistive Technology*, 19(3):604–622, 2024.

[Mob18] Zipf A. Francis L. Mobasheri, A. Openstreetmap data quality enrichment through awareness raising and collective action tools—experiences from a european project. 2018. Geo-Spatial Information Science, 21(3), 234–246. Accessed: 2025-06-10.

[Ope25a] OpenStreetMap Contributors. Openstreetmap, 2025. Accessed: 2025-04-13.

[Ope25b] OpenStreetMap Wiki Contributors. Key:access - openstreetmap wiki. https://wiki.openstreetmap.org/wiki/Key:access, 2025. Accessed: 2025-06-13.

[Pal25a] Pallets Projects. Flask documentation (2.2.3). https://flask.palletsprojects.com/, 2025. Accessed: 2025-06-10.

[Pal25b] Pallets Projects. Flask-sqlalchemy documentation. https://flask-sqlalchemy.

palletsprojects.com/, 2025. Accessed: 2025-06-10.

[Ron] Armin Ronacher.

[Sai25] Ali-Akber Saifee. Flask-limiter documentation. https://flask-limiter.readthedocs.io/, 2025. Accessed: 2025-06-14.

[Smu25] Simona Smultea. Guiding systems for people with visual impairment, 2025. Accessed: 2025-05-12.

[Soz25] Sozialhelden e.V. Wheelmap.org - find wheelchair accessible places. https://wheelmap.org/, 2025. Accessed: 2025-06-09.

[SQL25] SQLAlchemy Authors. Sqlalchemy - the python sql toolkit and object relational mapper. https://www.sqlalchemy.org/, 2025. Accessed: 2025-06-16.

[SV12] HENRI SIMULA and MERVI VUORI. Benefits and barriers of crowdsourcing in b2b firms: Generating ideas with internal and external crowds. 2012.

[The17] The World Bank. Romania: Children in public care. Report 117869, The World Bank, 2017. Accessed: 2025-04-10.

[Uni06] United Nations. Convention on the rights of persons with disabilities and optional protocol. Official Document, 2006. Adopted by General Assembly resolution 61/106 of 13 December 2006. Entered into force on 3 May 2008. Accessed: 2025-04-10.

[Vla25] Vladimir Agafonkin and Leaflet Contributors. Leaflet - api reference, 2025. Accessed: 2025-04-12.

[W3C25a] W3C Web Accessibility Initiative (WAI). Wai-aria overview. https://www.w3.org/WAI/standards-guidelines/aria/, 2025. Accessed: 2025-06-15.

[W3C25b] W3C Web Accessibility Initiative (WAI). Web content accessibility guidelines (wcag) overview, 2025. Accessed: 2025-04-14.

[YJ25] Hsiaoming Yang and Dan Jacob. Flask-wtf documentation. https://flask-wtf.readthedocs.io/, 2025. Accessed: 2025-06-10.