Departamentul Automatică şi Informatică Industrială
Facultatea Automatică şi Calculatoare
Universitatea POLITEHNICA din Bucureşti

LUCRARE DE LICENŢĂ

Web Application for Environmental Zones Awareness and Route Planning

Coordonator
Prof. Anca Daniela IONIȚĂ

Absolvent
Andrei SĂRĂNDAN

2024

Contents

Part I - General Aspects

1. Introduction

The threats of global warming have become more prominent than ever before, especially with recent wildfires and heatwaves occurring across the globe. The World Wide Web represents without a question an indispensable tool during present times and it should, ideally, be focused on creating a better world for future generations.

The scope of this project is to present a web-based application that helps users comply to traffic regulations mandated by The European Union, primarily focusing on environmental zones. This web application seeks to raise awareness and trust in this scope, enabling users to make informed decisions when traveling and consider the environmental impacts of their vehicles. While it also provides general information about environmental zones, the application represents more than just a text repository or a presentation website. This software system includes modern technologies such as application programming interfaces (APIs), extensive database model and complex access validation algorithms, all available from the user's account. These features are designed to offer an interactive experience, by introducing the possibility to register personal vehicles and use them in the route planning features. All the complex logic, data sets and algorithms are hidden behind the dynamic user interface, easily accessible from any web browser.

The development of this software system consists of three main parts. Firstly, the research on this topic highlights a number of aspects which are then improved by this application. For example, it covers the different complexities of multiple environmental zones throughout Europe. A key factor is to fit their evolving character, so that the information presented on the web application would be correct, up-to-date and organised in a concise manner. Secondly, developing a web-based application offers a number of advantages, such as accessibility, scalability and various technological resources. Different technical aspects must be taken into consideration, such as splitting the development between backend and frontend parts. This involves different programming languages or frameworks that must ensure communication between the user and the system. Finally, the extensive relational database model was designed to store relevant information about the environmental zones and the users. Since the legislations for environmental zones will change in the future, the database system was designed to cover the ones effective as of July 2024. However, the system supports future updates in order to provide relevant information.

As highlighted in the following chapters, the diverse range of regulations imposed in Europe may raise issues for individuals willing to travel. The development of this application is meant to solve these complexities and to help improve air quality by promoting environmental zones.

2. Inclusion of The Study's Domain of Interest

2.1. Environmental Zones

The concerns regarding increasing pollution necessitate the implementation of various measures aimed at enhancing air quality. Pollution, caused by multiple sources and presented in different forms, poses significant threats to both human health and the environment. Vehicular emissions stand out as one of the most prominent forms of pollution. This is mostly troubling in the urban settlements and metropolises that are suffocated by the raising number of cars, with domestic transport being one of the biggest growing pollution sectors since 1990 [1]. Out of all means of transportation, cars represent the most used vehicle, accounting for 60.6% of the total

emissions, with an average of 1.6 cars per European habitant. The European Union is striving to drastically reduce air pollution by implementing new legislations. Pushing new vehicles towards achieving zero CO_2 emissions is the primary goal. However, the actual progress has proven to be much slower than it was hoped.

The main methods of reducing CO_2 emissions from vehicles are manufacturing more efficient cars and changing the fuel type. This means transitioning to more eco-friendly ones, ideally obtained from green or regenerative energy [1]. Even though electric and hybrid cars are taking a considerable part of the newly registered vehicles, the issue of older, higher-emission vehicles must not be overlooked. Designing a schema to reduce the pollution proves to be a challenging, yet necessary aspect. Such schemas, referred as Urban Vehicle Access Regulations [2] or environmental zones, have been implemented in many Western European cities. These rules are found in different forms such as Low Emission Zones, Zero Emission Zones, tolls for driving in congested areas, changed parking rules and limited traffic in certain areas. The aim is to reach better air quality standards, while also improving traffic.

The European Commission categorizes vehicles based on the automobile class they correspond to, the fuel used for propulsion and emission class (Euro emission standard). Additionally, such regulations usually impose some form of toll or fee that must be paid in order to gain access into a specific zone. Out of the many existing forms of Urban Vehicle Access Regulations, most of them are Low Emission Zones (LEZs), accounting for 73% of the current regulations [2].

2.2. Low Emission Zones

The concept of environmental zones originated in Sweden in 1996 and was initially created to reduce both air pollution and large vehicles noise. These zones were eventually replaced by national regulations that comply with EU environmental standards and now only cover heavy-duty vehicles [3]. Such examples were later adopted by other countries and are now widely spread throughout Western Europe.

Low Emission Zones are geographical areas, usually located in large urban settlements, where specific restrictions have been imposed on more polluting vehicles. Typically, this kind of vehicles are prohibited from entering the zone, although in certain instances, a fee can be paid for access. Low Emission zones have proved to be a great method of reducing air pollution, especially targeting fine particles, which are highly correlated to several respiratory diseases.

In Europe, the two most concerning air pollutants are represented by nitrogen dioxide (NO_2) and particulate matter (PM, especially PM_{10} or $PM_{2.5}$ depending on the particle diameter). According to The European Environment Agency, these particles are responsible for a large number of premature deaths or chronic illnesses [4][5]. By restricting the access of highly polluting vehicles in central areas of urban settlements, authorities can diminish the pollution that affects the environment and the health of the citizens. Another approach would be retrofitting older vehicles to meet modern regulations, by adding diesel particle filters (DPF). These filters could elevate the cars to a slightly higher emission standard [6].

The European countries plan to gradually implement such environmental areas in the following years [5]. This incremental approach allows each nation to shape their legal framework to fit the needs and possibilities of the country and its citizens. Over more, in some cases, the vehicle access requirements are set by the municipal authority, following the national legislation. Depending on the national law, the Low Emission Zone term can be found under different names.

2.3. Study of Official Rules Regarding Environmental Zones in Europe

2.3.1. Germany and France

In Germany, Low Emission Zones are known as "Umweltzonen", while in France they are referred as ZCR ("Zone à Circulation Restreinte"). Both terms define strict legislation that separates vehicles based on Euro standard. A sticker is assigned to each vehicle and must be displayed on the windshield to be visible for authorities. Even though the legislations in these countries seem similar, they are actually quite different.

For example, one of the main differences is represented by the classification criteria. In Germany, vehicles can be assigned one of the three available emission stickers. Green represents the cleanest and most environmentally friendly out of them all, followed by yellow and, lastly, red for the highest polluting vehicle. These stickers were first introduced in 2008 and because of the regulations in place at that time, a diesel vehicle now needs to be at least EURO 4 to obtain the green sticker [7].

On the other hand, France introduced the Crit'Air vignette system in 2017. Crit'Air stickers are also color-coded and classify vehicles based on emissions. However, France has designated six categories ranging from green, which is only assigned to 100% electric or hydrogen vehicles, to dark grey for the most polluting vehicles. The Crit'Air vignette is part of a long-term strategy to introduce environmental zones throughout the cities with more than 150.000 inhabitants [8]. If the air quality thresholds are not met, authorities plan to raise the minimum access criteria by imposing a higher-class registration. As an example, in Paris, regulations in place since 1st of June 2021 impose minimum Crit'Air sticker 3 (diesel EURO 4 and EURO 2 petrol cars). Starting with the year of 2025, Crit'Air sticker 2 (minimum diesel EURO 5 and EURO 4 petrol cars) will be required. Initially, this restriction was scheduled to be implemented in 2023, but it had to be postponed [9].

2.3.2. Italy and Spain

Italy and Spain are two popular destinations for tourists in Western Europe. Generally, they have many things in common, however when it comes to environmental regulations, the frameworks are very different.

Spain selected sticker-based registrations, which are also colour coded and somewhat similar to those in France and Germany. These registrations are named Distintivo-Ambiental and must be displayed in the windshield in order to highlight the emission class of the vehicle. However, these stickers are only available for nationally registered vehicles. In contrast to other frameworks, Spain officially recognizes the sticker registrations issued by other countries as long as they comply with the standards imposed by the Spanish LEZ. Recently, Barcelona introduced an additional digital registration which is required for foreign vehicles. This registration can be issued for vehicles that are in the same (or higher) emission class as the required Distintivo-Ambiental. Momentarily this is B-class sticker, which corresponds to Euro 3 petrol or Euro 4 and 5 diesel vehicles.

Italy has the largest number of environmental zones in Europe. Here, the legal framework is different and no registration is required for entry. A notable mention is that the regulations are imposed separately and vary from one region to another. In contrast to most countries, Italy also introduces different restrictions during the winter period [10]. These are typically stricter than the ones that apply during the rest of the year, in order to avoid the smog effect. Further, these

restrictions are not permanent (365 days/year) like in most countries. Evenings, early morning and some holidays are generally excepted from vehicle restrictions.

2.3.3. Northern Europe

Countries in this region implement their own systems with the purpose of regulating vehicle emissions. For example, only Denmark targets regular cars in Low Emission Zones, also referred as Miljøzone. The rest of them primarily focus on heavy vehicles. Even though the scope remains the same, mainly to reduce pollution by restricting the access of highly polluting vehicles, the minimum requirements and access validation are different.

Low Emission Zones in Northern Europe vary from those described above. First of all, sticker-based registrations do not exist and foreign ones are not accepted. Each nation sets the minimum requirement for combustion-based vehicles and those willing to drive through must comply. Within these frameworks, vehicles registered locally are automatically included in the central national database, which considerably facilitates the access for citizens. However, foreign vehicles need to register online, typically prior to entering the LEZ. Otherwise, even if the vehicle could be eligible for access, the authorities impose penalties. These restrictions are primarily enforced by smart-camera systems, which capture the vehicle's registration number when entering the designated areas and verify the vehicle's eligibility in the central database. This automated approach enhances efficiency and accuracy, while reducing the costs associated with manual inspections.

2.3.4. Eastern Europe

Poland authorities encourage the population to use newer and less polluting vehicles. Their environmental restrictions also consider the last registration date of the vehicle. As of 2024 there are two cities (Warsaw and Kraków) that adopt this strategy, with a third city planning to join the list at the beginning of the following year. Each of them imposes different restrictions, based on Euro emission standard and fuel type, but unlike other countries, here the restrictions are especially strict for vehicles registered after March 1st, 2023. By doing so, the older vehicles which had already been registered are still allowed in traffic, but acquiring a highly polluting vehicle is discouraged.

Hungarian framework mainly restricts the access of heavy vehicles. One additional pollution schema is notable for the capital, Budapest. Here, in normal circumstances, access is not restricted for regular vehicles. However, if the pollution raises to alarming rates, the capital instates a temporary environmental zone that restricts access for vehicles under Euro 5.

One country that has recently made steps towards reducing CO2 emission in central areas is Bulgaria. The framework here is still new and as of March 2024, no penalties have been imposed. The capital Sofia is the only city that has such restrictions and only during the winter periods. The national framework provides nationally registered vehicles with a sticker that reflects the emission class. Over time, the plan is to gradually raise the standard, by introducing new stickers and extend the area to other cities.

2.4. Forecasts

According to the report made by statista.com [11], forecasts expect a significant expansion of Low Emission Zones throughout Europe. Projections indicate a substantial increase in the number of LEZs, with an estimated total of 510 zones across seventeen countries. This increase

highlights the European Union's commitment to mitigating air pollution and promoting sustainable urban development throughout the continent.

Country	Number of LEZ in 2022	Project number of LEZs by 2025
Italy	172	172
Germany	78	78
United Kingdom	17	18
Netherlands	14	14
France	8	42
Sweden	8	8
Austria	6	6
Denmark	4	4
Spain	3	149
Belgium	3	4
Norway	3	3
Czechia	1	1
Finland	1	1
Greece	1	1
Portugal	1	1
Poland	0	2
Bulgaria	0	3

Tabel 1. Number of Low-Emissions Zones in Europe 2022-2025[9]

2.5. Analysis of The Heterogenous Situation in Europe

The scenarios from the previous chapter only describe some of the twelve countries covered by the application presented in this document. They underline the difficulty of understanding and following the regulations throughout Europe. In this context, travelling becomes a significant challenge for the average individual.

The European Union encourages member states to regulate vehicle access in urban settlements. When doing this, national authorities must consider different factors such as population density, infrastructure or public transport. This non-homogeneous structure offers several advantages. Having diverse strategies creates the possibility to evaluate the efficiency of different models concurrently. It is important to acknowledge that governments are best positioned to identify the environmental challenges in their countries and impose efficient solutions accordingly. For example, as The Deputy Mayor for Transport in United Kingdom has confirmed in an interview [12], the automated monitoring systems, such as plate recognition cameras, prove to be an effective enforcement strategy.

However, this system can become very confusing for the population because of all the different legislations. With regulations varying not only between countries, but also among different municipalities of the same country, drivers may encounter difficulties. The Crit'Air sticker system in France and the Umweltplakette system in Germany are a good example. These two countries alone impose nine stickers, each of them available exclusively within its own jurisdiction. The only similarity between the two schemas is the fact that they use color-coded sticker registrations. The language barrier can also pose challenges for most people. From "Umweltzonen" in German, to " Miljøzone " in Danish, and " Distintivo Ambiental" in Spanish, the list goes on for each of the seventeen countries.

The effectiveness of Low Emission Zones is without a doubt linked to the diligence of the framework. But, nevertheless, in order to be effective, the majority of the population must comply to these regulations. While the establishment of Low Emission Zones represents a proactive first step towards improving air quality, in reality, their impact relies on the degree individuals approve and respect this concept. Even the most restrictive LEZ strategy can fail to reach the intended goal if control measures are lacking or if vehicle owners do not comply.

Environmental zones are not yet embraced by all individuals, therefore, public awareness campaigns can serve a crucial role. The European countries must convince the population that environmental zones are not meant to impose financial burdens, but rather to mitigate pollution in highly populated urban centres. Such initiatives are part of a long-term plan designed to achieve lasting benefits for public health and environment.

2.6. Available Applications

2.6.1. Green-Zones.eu

The Green Zones application is mobile based and represents a valuable resource for individuals seeking to understand and comply to the environmental regulations in Europe. Here, users have the possibility to register their vehicle and to check compliance with environmental zones. The disadvantage is that these features are limited for users that do not pay a subscription. For example, regular users can verify if the vehicle is eligible for access in a city but cannot use the search bar to find the actual location. This, along with many other features such as registering more vehicles, are only available to premium users.

The subscription-based model requires a monthly fee and is rather suitable for enterprise users. For regular users, who only plan a few trips each year, the associated fee could cause significant drawbacks. Moreover, the application only promises up to date information for users with an active subscription plan. In most cases, the costs may outweigh the benefits for individuals.

Additionally, Green Zones offers a dedicated website that complements the mobile application. Here users can find further information on different environmental zones, including LEZs. Still, the information is not guaranteed to be up to date. The website can also be used to purchase registrations or vignettes for several countries action which cannot be done from the mobile application. Therefore, the mobile application is not enough on its own.

Overall, the Green Zones application offers a user-friendly interface and valuable features for navigating LEZs, thereby promoting environmental awareness. However, the subscription-based model and the split between the two platforms represent entry barriers for some users.

2.6.2. Urban Regulations

UrbanAccessRegulations.eu represents an online platform that supports information about different traffic restrictions throughout Europe. This website is easily accessible from any web browser. It includes the complete range of regulations, from Low Emission Zones to road tolls and emergency schemas. The website follows a cost-free model, but packs all the information in a relatively large number of pages.

Even though it drops financial burdens, it does not fully facilitate the access to information because of the way it is displayed. Over more, in some cases, the information is not up to date. The blog or text approach does not shine in this context, as modern technologies offer better methods of delivering the information in online applications. Here, each country has a dedicated web page, presenting all the traffic regulations. Then, the information is segregated in different pages for each

city. Users need to spend a lot of time before finding the details relevant to their needs. An interesting approach would be manipulating all this data based on the user's requests.

Substantially, UrbanAccessRegulations.eu provides resources for users seeking information on vehicle regulations throughout Europe. While the website compounds comprehensive coverage and is free to use, the amount of information and less user-friendly navigation may pose challenges for those unfamiliar with the topic.

3. Technical Documentation

3.1. Software Technologies

3.1.1. Hypertext Markup Language

Markup languages are commonly used in various applications, from web development to document processing and data transfer. Generally, their purpose is to provide a standardized approach of representing and transferring data, thereby enabling communication between different platforms and software systems. Some of the well-known languages are HTML (Hypertext Markup Language) and XML (Extensible Markup Language). The last one is widely used for data storage, document parsing and configuration files.

Hypertext Markup Language (HTML) serves as the core foundation of web application development and is used to create the structure and content of the web pages. HTML syntax is structured based on elements created by tags and key words, as it is common for markup languages. The tags are enclosed within angle brackets (</>) and contain attributes which specify additional characteristics about the elements.

In web development, HTML is used to define the structure and layout of the page. By creating HTML files, developers compose functional documents based on specific elements such as divisions, paragraphs, spans, headings, lists, forms, images, anchor tags etc. By having a clear and concise syntax, HTML facilitates the process of creating well-structured web pages that can be interpreted by most web browsers, across different devices and operation systems. From an architectural perspective HTML can be viewed as the foundation of the web system. To improve the visual appearance and to create interactive web applications, HTML lays the possibility of integrating dynamic features built by other technologies, such as CSS and JavaScript.

HTML5 represents the latest evolution of the HTML standard, adding new modern features and capabilities designed to enhance the development of web applications. Some of the newer elements introduced are <*header*>, <*nav*>, <*article*> and <*footer*>. Using these tags provides a well-organized structure and a better-defined meaning of the web documents. Additionally, HTML5 offers support for multimedia elements through tag elements, such as <*video*> or <*audio*>, but also brings advanced controls for forms and APIs [13].

3.1.2. Python

Python is a powerful programming language known for its versatility and reliability, which makes it an excellent choice for a wide range of applications. The programming language is relatively old, but its best version, Python 3, was introduced in 2008 and improved many aspects of the previous iterations. Its expressive syntax is somehow similar to the English language, at least comparing to other programming languages. One notable advantage of Python is the extensive library and wide range of third-party packages. Through these, Python can tackle diverse tasks from data analysis and artificial intelligence to automation or even web development [14].

The flexibility allows developers to quickly evaluate their software and implement the fully functional products. The clean and intuitive syntax, combined with typing extensions, generally improve code readability and maintainability, making it accessible to both junior and senior programmers. Python benefits of automatic memory management which simplifies the development, reducing the time and resources required to build and manage complex software systems. Over more, most modern machines and operating systems, such as Windows or Linux, can run Python code.

When it comes to web development, Python offers several solid solutions for scalable backend mechanisms. The main frameworks for web development are Flask and Django, which provide the features necessary to create and manage secure web applications. Over more, Python's extended support for database integration and REST APIs makes it perfect for the backend development of complex software applications. The dedicated frameworks ensure seamless interaction with the frontend layers, including those built in other programming technologies.

Overall, because of its versatility and large ecosystem, Python represents a popular choice for backend development, offering users the tools and the freedom to create complex and reliable web applications. Its popularity in web development was highlighted by various surveys and rankings conducted by reputable organizations. For instance, the Stack Overflow Developer Survey [15] consistently ranks Python among the top programming languages, highlighting its widespread adoption and relevance in the industry.

3.1.3. Flask

Flask is a flexible, lightweight Python framework, efficient in web development as it allows software engineers to quickly implement the fundamentals of their applications. It was developed by Armin Ronacher and became popular for its minimalist design that provides features used to build full-scale web applications in a relatively short time. As highlighted by Miguel Grinberg in his book [16], Flask framework offers the essential tools to handle HTTP requests, configure URL endpoints, dynamically render web templates or fetch REST APIs, while also maintaining the security and scalability of the system.

Integrating the different technologies necessary in the development of a complex Flask web application implies a rigorous management of the file structure. This strategic approach ensures scalability and adaptability to future changes. Long term, these aspects are essential for system applications aligned with modern software standards. The minimalist design of Flask applications does not constrain developers with strict coding patterns and allows them to focus their resources on innovation and problem-solving.

For route handling features, Flask uses Werkzeug library. This library follows the WSGI protocol (Web Server Gateway Interface) in order to ensure interoperability between web applications. Concurrent user requests are managed by asynchronous software elements. By default, Flask launches each user request in a new thread and manages the processing based on the operation system specifications. This ensures efficient request handling and optimal performance even under heavy loads [16]. Over more, Flask's lightweight footprint and resource usage make it suitable for building APIs, microservices or other lightweight web applications, where resource efficiency is essential.

Flask applications are used in different industry sectors, from small businesses or startups to large enterprises. Because of the versatility and the short development time needed to construct a demo application, Flask represents a viable choice for many software engineers. While in some cases Flask is used to build Minimum Viable Products (MVPs), it is also found in production-

ready applications. Some of the most notable applications built with Flask include Netflix, Airbnb, or Uber [17] but it's also widely used in microblogging platforms, content management systems (CMS), REST API development and data analysis platforms. Flask's extensibility makes it suitable for projects of any size and complexity, offering developers the flexibility to gradually scale and evolve their applications when needed.

3.1.4. SQLite

SQLite is a self-contained, lightweight engine used for relational database management, that is widely acclaimed for its reliability and versatility. The term "self-contained" refers to the fact that databases are stored in a single file. Particularly, it is favoured for its seamless integration into various applications, with minimal required configuration and administrative effort. Despite the lightweight nature of SQLite, it offers powerful features and capabilities, commonly associated with more extensive database systems. SQLite provides capabilities for transactions management, indexes, and triggers. The database can be accessed using a custom variant of the classic SQL query language [18].

One notable aspect of SQLite is its support for object-relational mapping (ORM), a fundamental concept in object-oriented programming. This method maps classes to tables in the database of the application, creating a seamless interaction between codebase and the database [18]. Over more, popular third-party libraries, such as SQLAlchemy, provide advanced ORM features, further enhancing the flexibility and functionality of Python applications with SQLite databases.

Overall, SQLite's mix of simplicity and reliability makes the combination with Flask a viable choice for the development of small or medium web applications. The support for class modelling is seamlessly integrated with objected-oriented concepts, further solidifying its position as one of the most preferred database technologies. HG Insights statistics mark SQLite as a data management engine used in many large enterprises [20].

3.1.5. JavaScript

JavaScript (JS) is well renowned for its versatility and became one of the most important programming languages in modern web development. JavaScript powers a wide range of online experiences, from dynamic web pages to interactive web applications, making it an indispensable tool for web developers worldwide. For these reasons, JS is considered the backbone of web applications, being used by almost all online websites [22].

Generally, the syntax is considered lightweight and the code structure can be adapted to specific needs. Many IDEs offer good JavaScript integration and code autocomplete which contribute to the appeal of this programming language to software development community. The amount of code necessary for creating basic elements is relatively short, which allows developers to quickly explore different approaches and refine the code before releasing the final product. JavaScript programming language adheres to modern web standards while also ensuring cross-platform compatibility.

Over the years, a multitude of libraries and frameworks have emerged to further extend JavaScript's capabilities. Among the most popular frontend frameworks, React.js, maintained by Meta, has gained widespread adoption for its component-based architecture and efficient Document Object Model manipulation. On the other hand, Angular, maintained by Google, is also a great tool, suitable for developing small or single-page applications. It offers features like dependency injection and two-way data binding to facilitate the creation of scalable and

maintainable web applications. Working with the extensive ecosystem of libraries and plugins surrounding JavaScript further amplifies its capabilities, providing a bundle of pre-built solutions for common development tasks.

Nevertheless, "the vanilla" version of the language (the core functionalities of JavaScript, without libraries or frameworks) still remains a solid option for many applications. Its complex features, yet flexible approach, allows the creation of customized solutions without the need for additional dependencies, promoting efficient development practices.

JavaScript shines through its seamless integration with various APIs. This way, developers can access a wide range of functionalities within their applications. Whether it is integrating external or third-party API services, consuming REST APIs from the backend server or using the browser to manipulate the Document Object Model, JavaScript offers robust solutions for building interactive web applications.

Asynchronous programming [22] represents another standout feature of JavaScript, allowing non-blocking operations and real-time updates of the web page, thereby enhancing application responsiveness. Through mechanisms such as async functions, that use "promises" and "async/await" methods, JavaScript succeeds in managing asynchronous tasks. This programming technique ensures a smooth user experience even when completing complex tasks and loading larger sets of data.

JavaScript is also commonly used in combination with backend frameworks from different programming languages, like Flask. In these cases, JS serves an essential role in the bidirectional communication between the frontend and backend components of a web application. Through programming techniques like AJAX (Asynchronous JavaScript and XML), JavaScript facilitates fast transfer of data.

The architecture of JavaScript is event-driven and creates the possibility to handle multiple requests simultaneously, further enhancing the intuitive user interfaces. Event listeners attached to different elements of the web page ensure a response from the system to the user interactions.

In summary, JavaScript can be suitable in both frontend and backend web development, as it represents an essential technology in the modern web ecosystem. The elegant syntax, cross-platform compatibility and extensive ecosystem, place JavaScript as one of the most popular tools for robust, interactive, and scalable web applications.

3.1.6. Application Programming Interfaces

Application Programming Interfaces (APIs) are sets of rules or mechanisms created in programming code which function as intermediaries for different software applications. These rules define the methods through which different software components can interact with each other [23]. All sides involved in the communication follow the predefined protocols to request and send data. APIs are essential in contemporary software engineering, as they are used by developers to integrate different components, to connect a multitude of services, tools, and functionalities.

One of the key advantages that comes with using APIs is the abstraction of complex processes. Software engineers can focus on developing or importing external features without the necessity to go through all the functionalities of those features. Additionally, APIs are used to create modular and scalable software architectures. Breaking down large systems into smaller components makes them easier to develop and manage. With this modular approach, the strategy is to build individual components and interconnect them through APIs [23].

3.1.7. UML

UML stands for Unified Model Language and represents a standardized set of diagrams which are meant to help software engineers and stakeholders develop and visualise the system. This modelling language offers blueprints for a wide range of infrastructures, which reflect different perspectives over the system [24]. These diagrams can be viewed both from a technical or from a business perspective. UML includes structural diagrams to represent the static aspects, behavioural diagrams to represent the dynamic aspects and additional diagrams for the interactions between elements of the system.

The main advantage of using a standardized modelling language is that all stakeholders can get an understanding of the project, regardless of their technical background. The easy-to-read nature of UML diagrams makes them widely used in project documentations.

The industry offers plenty of tools for developers to create UML diagrams, which are usually integrated during the analysis and design phases of the system's lifecycle. The features of UML are extended by other design languages, such as SysML (Systems Modelling Language), which was specifically designed to include complex hardware and software elements used in system engineering.

3.2. Backend Development

Backend development represents the server-side component of web applications that is responsible for most critical activities, such as logic implementation, data processing, and handling communication with the user interface. A strong backend is essential for complex and large-scale applications, empowering them to manage data effectively. Without the backend algorithms, web applications are essentially limited to static website pages, lacking the dynamic functionalities available in most modern digital experiences [25] [26].

In web applications, one important aspect is the configuration of URL routes. This defines the paths that users follow to access different parts of the application. The route handler functions act as pathways for directing web requests to the appropriate resources within the application. Route handling dictates how URLs are mapped to specific functions within the application. In Flask applications routes are generally managed within the *views.py* file, where they are defined by special functions [27].

Generally, web systems include multiple HTML files that are managed in the backend. Modern technologies include tools that configurate dynamic URL endpoints, where the backend processes the data rendered in the interface. Typically, this is achieved by using a set of predefined functions and algorithms. The system can set up a single endpoint function to handle multiple requests (POST, GET etc.), and depending on the URL, it may return different resources. This approach reduces redundant code and enable centralised control from the backend engine. A common technique is to pass user input as an argument in the handler function, based on which the server determines the result.

3.3. Frontend Development

Frontend Layer represents the graphic interface used by the client to operate the system. In web applications, this interface is rendered by the user's web browser and should prioritize both visual and intuitive appeals. This is important as most users or stakeholders do not have technical background. Web applications use the frontend layer to hide the various technical complexities from the user. Modern applications set high standards for the frontend component, as user interfaces should be responsive enough to ensure real time communication with the system [25].

Primarily, the web application is composed of different HTML files, structurally linked between them. While the HTML allows the creation of simple web pages, without the added features of CSS (Cascading Style Sheets) and other programming languages such as JavaScript, the design is plain and their functionality is static, limited to predefined text. From an architecture perspective, HTML templates can be viewed as the foundation of the application, using the tag elements as bricks. On top of that, CSS adds the design and JavaScript sets the functionality and interconnects the elements. Typically, Flask application store these HTML files inside the *templates* folder, while CSS and JavaScript files are stored inside the *static* folder [27]. Following the protocol or the rules of structuring files and directories is just as important as following the coding standards when creating a complex software application. This is especially relevant for web applications, which often use a combination of different programming languages and file types.

HTML files contain the head element (<*head*>) where they link static files. This linking allows elements defined in the HTML templates to be modified by the software resources from the *static* folder. It is important that the HTML tags contain *id* or *class* attributes. They can be used by identifiers, both in the stylesheets (CSS files) or in the JavaScript files. While the ID is unique and refers to a particular element, the *class* attribute refers to a bunch of items that have the same design or behaviour. Notably, one HTML element can have several *class* attributes [13].

3.4. External APIs

3.4.1. Google Maps API

While highlighting the different LEZs can be done through various approaches, in this case, the application uses dynamically rendered maps that were created with Maps API. Out of the long list of Google services, Maps JavaScript API stands out as it fits perfectly to the requirements of this web application. The API allows creation and web integration of custom, dynamic and interactive maps. Setting up a map element is fairly simple and developers can use the online documentation provided by Google. Basically, the map is an HTML <*div*> element, customized with JavaScript. In order to use the API, developers must register on the official website and get an API key. This key represents the credentials of the user and must be included in the JavaScript code that makes the API requests [28].

3.4.2. OpenCage Geocoding API

The geocoding technique represents the conversion of a location or an address to geographical coordinates (or reverse) expressed in latitude and longitude. OpenCage offers great solutions for the forward or reverse geocoding process in various programming languages. The tool represents a viable alternative for the Geocoding API offered by Google, as it drastically reduces the cost. This is ideal for applications that handle a large number of API requests [29].

3.4.3. Air Quality API

Air quality is monitored and scored by the World Air Quality Index (AQI) [30]. It measures the particle matter and carbon emission using multiple GAIA monitoring stations placed around cities. Based on the data gathered by these sensors, the AQI scores a number between 0-300+. Generally, air quality score is considered good when the AQI is below 50 and acceptable when the AQI is below 100.

The API offered by the [30] project allows developers to integrate AQI data in their application through different approaches, such as creating widgets or displaying markers on maps generated by other API providers.

3.5. Software Development Tools

3.5.1. Integrated Development Environment (IDE)

Integrated Development Environment represents one of the most basic tools for any software developer. They extend the functionality of writing programming code from plain text, by adding features for compiling and debugging. They are ideal for extensive coding sessions, as they improve programmer efficiency, offering code auto-completion or syntax suggestions based on the selected programming language [31]. For big projects, like the development of a fully functional web application, where multiple programming languages are used and shared among several code files, the IDEs help developers easily store, navigate, and organize the program files.

Visual Studio Codes was developed by Microsoft and perfectly integrates the features of IDEs described above. The lightweight feel and flexibility helped VS Code become one of the most popular IDEs on the market. First of all, VS Code ensures smooth and short load times across multiple platforms, compared to other IDEs. The integrated terminal allows quick debugging, commands and scripts. Additionally, VS Code provides extensions for Git integration, used in version management, or ones for live web server. It also perfectly integrates Python language, with the possibility to install libraries or frameworks using the pip command directly in the command line.

3.5.2. Virtual Environments

Since the application includes a multitude of Python libraries which need to be imported and installed on the device that runs the server, the management can be optimised by creating a dedicated virtual environment. One of the very first steps in the software development lifecycle is represented by the setup of a new virtual environment. In many cases, this is done through a dedicated software.

Anaconda is a software application designed for package and environment management, specifically for Python and R programming language [32]. This allows developers to create different virtual environments on the same device. Each environment represents a separate and isolated workspace that contains the software required to compile and execute the application code. By isolating the components, developers manage the dependencies between the different libraries more efficiently. Separating the projects to specific virtual environments ensures that the applications benefit of the required libraries, while also avoiding conflicting packages or versions. This approach is crucial in modern software development, as it allows easy bug fixing, ensures the global environment of the host does not get corrupt from conflicting libraries and also ensures smooth upgrades.

Anaconda Prompt and Anaconda Navigator are two of the tools used for creating and managing virtual environments. The first of them is a command-line interface which allows developers to create, manage or activate virtual environments with short keyboard inputs. On the other hand, the Navigator offers a guided user interface which simplifies the process of managing and understanding the settings and parameters of virtual environments. The integration of anaconda with Visual Studio Code ensures smooth and optimal setup of the libraries required in the software development process.

3.5.3. Database Management Tools

While the command line integrated in Visual Studio Code offers the possibility of opening the database file and executing SQL commands, this option lacks the versatility and the visualisation aspects offered by other dedicated software applications. For smooth and optimal modelling of the database, the development process may use DB Browser, an application dedicated to managing SQLite database files. This open-source tool offers a graphic interface used for database visualisation. Additionally, developers can edit the tables, use the search function for quick data visualisation, but can also execute SQL commands and queries in the dedicated terminal.

3.6. Web Development Techniques

3.6.1. REST APIs

A Web API, also known as a Web Service API, serves as the communication protocol between the web server and the browser. In this case, the interface conducts the communication and data exchange between them. In general, all web services are considered APIs, however, not all APIs are specifically designed for web systems. Among the different types of APIs, the REST APIs represent the commonly used ones in web development. In fact, they have become so widespread that in the context of modern web development, the term API alone is often used for referring to REST APIs specifically. This monopole over the industry reflects the shift towards web-based architectures [33].

Representational State Transfer Application Programming Interfaces (REST APIs) ensure the communication between the caller and the provider of a resource. They are frequently used in systems based on the Client-Server model. While in general APIs are considered a set of rules and protocols that manage the communication between two parties, in order for an API to be considered RESTful, it must follow additional characteristics.

REST APIs facilitate online communication via HTTP requests, which are used to perform standard database operations such as creating, reading, updating, and deleting records. They correspond to the well-known HTTP methods: GET, POST, PUT, DELETE etc. Data delivered through these requests is structured in various formats including JSON, XML or plain text. Through REST APIs, the system transfers a representation of the resource's state between the server and the client. The endpoint refers to a specific URL, accessed by the consumer of the resource, with the scope of performing some action or retrieving information [34].

The communication is considered stateless, as each request is independent and the GET requests do not store client information. REST APIs are not dependent to the programming language, rather to the principles that stand behind the implementation. Resources managed by the REST APIs should be assigned to only one endpoint and requests must contain all the necessary information. Important information, such as authorizations metadata and uniform resource identifiers (URIs), is included in the HTTP request header [33][34].

3.6.2. CRUD Operations

CRUD operations are fundamental in web applications and are often used in RESTful APIs. CRUD stands for Create, Read, Update, and Delete. These are the four basic operations that can be performed on a resource of a database. They are critical for maintaining the integrity of databases as they ensure that data can be added, retrieved, modified. Such features must be

conducted in a controlled and predictable manner, in order to ensure the development and operation of software systems [33][34].

In Flask, these operations can be implemented using Flask's request handling, and SQLAlchemy for ORM (Object-Relational Mapping). The implementation involves defining routes for handling requests to perform database operations. Each URL endpoint provides a method for handling HTTP requests. This is how the users interact with the web system. SQLAlchemy makes the process of interacting with the database straightforward, ensuring that data management is efficient and organized. Integrating the CRUD operations ensures controlled flow of data.

3.6.3. JSON

JSON is the abbreviation for JavaScript Object Notion and despite the name, this format is independent of the programming language, as it is seamlessly used in C/C++/C#, Java or Python. The JSON format is a lightweight method of transferring data, as it is interpreted by most machines. Developers can also easily understand information stored in JSON because its structure is similar to plain text or other elements familiar to most developers.

This format structures data as a collection of key and value pairs, similar to dictionaries in Python or other programming languages. The keys are always strings and the values can be of different data types, including arrays[35]. In short, JSON formats data similarly to JavaScript objects, therefore converting from JSON to JS objects becomes extremely simple. This is one of the reasons why JSON format is preferred over other alternatives, such as XML. The simple syntax ensures interoperability, making JSON one of the most frequently used data formats for transferring data online.

3.6.4. AJAX

AJAX represents a set of techniques used in web development and is short for Asynchronous JavaScript and XML. It allows the client-server communication to take place without the need to reload the page after every HTTP request. This is achieved because the client and the server only exchange small packages of data at a time. AJAX technique fluidifies the experience, by minimizing interruptions, and allows instant updates of the HTML page. AJAX includes technologies like Document Object Model, JavaScript or XMLHttpRequests. These make AJAX heavily dependent to the browser used by the client, as the JavaScript is interpreted by the web browser and some of them do not offer support for these techniques [36].

Traditionally, in web applications, the user interaction is locked during the submission process of the HTTP request. AJAX sends and receives data from the server asynchronously, handling the communication without interruptions. Users do not have to wait for the HTTP request to be completed for the interface to be updated dynamically. This update is possible without fully reloading the web page.

Asynchronous programming is a key feature of JavaScript uses special functions. These functions use the await key word to suspend execution until specified conditions are fulfilled. This behaviour is called promise-based, because async functions return Promise objects. Based on their value, conditional programming is executed with structures like try-catch.

3.6.5. Document Object Model

Document Object Model (DOM) structures documents as logical trees, consisting of nodes and objects. In this technique, nodes contain objects with event handlers attached and are

interconnected by the branches of the tree. Using the DOM method, the tree elements can be accessed by software engineers in order to change the content or the structure of the document [36]. DOM offers different types of nodes for documents, elements, text, attributes, or comments. Each node represents an HTML element.

Objects, on the other hand, are instances of classes which define the structure and behaviour of the document. In JavaScript, these objects can be accessed, by methods such as *getElementByID(),* and manipulated, by methods such as *appendChild()*or *setAttribute()*.

By integrating the DOM and AJAX with JavaScript programming language, the frontend layer of the application fluidifies the user experience. The combination of the two techniques ensures the dynamic and responsive nature of the application. The graphic interface allows users to directly input data in the document model and obtain instant feedback. In fact, the data is sent to the backend by the HTML requests, which are handled by the asynchronous communication functions. Based on the response generated in the backend layer, the JavaScript algorithms render the frontend interface by manipulating the DOM to offer real time update.

3.6.6. Object-Relational Mapping

While the Object-Relational Mapping (ORM) was first introduced in Java in the early 2000s, the technique is now widely used across several programming languages. Python offers the possibility to integrate ORM techniques in the management of the database systems [37]. One of the most popular ORM libraries is SQL Alchemy. The ORM techniques create an abstraction layer, that maps Object-Oriented elements of the programming language to relational database elements [38]. More specifically, this procedure maps classes to database tables, and objects to rows or entries in those tables. Once the fields of the tables are populated, developers can access them using object-oriented methods along with executing SQL queries.

SQL Alchemy is a very popular ORM library and is often used in the management of SQLite databases. In Flask, application tables are created by classes that are configured in the *models.py* file. This approach puts the focus on data processing and business logic, concentrating the code required to interact with the database in the backend layer. Classes mapped to tables can be easily adapted to fit future updates, guaranteeing easy maintenance of the database schema.

3.6.7. Authentication and Authorization

Creating an account allows the user to personalize his experience and interaction with the application, as it facilitates the access to storing information in the database. Authentication refers to the verification of the user's identity, by comparing the credentials used for login to those stored in the database. Obviously, the user needs to have an already created account, therefore using the credentials saved in the database.

The authorization determines the actions available to the user, usually based on his role. This aspect is critical for the security of the entire system [39].

3.6.8. Templating Engine

Template engines represent tools used to generate dynamic HTML pages, based on a static template. This template contains placeholders for data and serves as blueprint for other files. On runtime, or when necessary, the placeholders are filled in with dynamic data, usually extracted from the database. This manages some HTML files in the backend of the application, promoting cleaner and maintainable code [16].

Jinja is a powerful templating tool for Python and can be integrated with several web frameworks. It is best used in combination with Flask, as Jinja represents the default templating engine. The Jinja web template is an HTML file that also contains specific variables, functions and tags which help the implementation of programming logic. The syntax is similar to classic Python, as it's been derived from it. Jinja offers the possibility to create inheritance relationships between HTML files. This means that the base template, usually named accordingly as *base.html,* is extended by other HTML files, called child templates. The base file acts as the backbone for the other files, as through extension, just like in Object Oriented Programming, all the variables and functions are passed to the children. The extension includes CSS or JavaScript files, linked in the base file, or even entire HTML structures, which reduce the amount of necessary code [39].

PART II – PROJECT CONTRIBUTIONS

4. Project Motivation

4.1. The Complexity of Navigation Throughout Environmental Zones

Environmental zones are not only beneficial, but essential to the well-being of our society and future generations. Efforts must be made in order to reduce pollution, as the effects of global warming cannot be neglected or denied. As time passes, more and more European countries adopt legal frameworks to regulate the access of highly polluting vehicles. In the past few years, hundreds of cities have adopted different variations of environmental zones that restrict CO_2 emissions in urban settlements.

The wide-spread presence of the environmental zones is a big step towards mitigating pollution in the urban areas. The main problem is represented by the differences between the legislations imposed in European countries. Some people may not be affected by these restrictions if they do not travel often or if they do not use personal vehicles. However, many citizens are confused by the different legislations they must comply to. The issue in itself is not the high density of the zones, but the fact that each environmental zone has different regulations. This is still an emerging topic, which is going to change over time, and presumably a definitive schema might be adopted throughout all Europe. But this is just a prediction and the navigation may become even more complicated.

It is even more concerning that available resources often fall short in providing clear details about the various restrictions that people must comply to. While online information is indeed available, usually it is dispersed across many different platforms, with segmentation made by countries or even by cities. Fragmented landscapes can be overwhelming for regular users, who would often find themselves spending significant time navigating through multiple websites when gathering relevant information for their specific situation and needs. Changing the legal framework in all European countries is an impossible task. Convincing the population to act in a way that is favourable to themselves and the environment is a more realistic goal. However, this is indeed a challenging aspect which has yet to be accomplished.

At the moment, navigating through environmental zones in Europe is more difficult than ever before. For vehicle restrictions to be effective, efforts need to be made by both the population and official authorities. Unfortunately, in many cases, if the information is not accessible or concise enough for regular individuals, the restrictions imposed in these areas will be violated. In this case, even if authorities impose further penalties, the scope of the environmental areas has failed, because the pollution is still on the raise.

4.2. The Recommended Solution

This project composes a solution for reducing the complexity and confusion created by the different European Low Emission Zone schemas. This represents the development of a new web-based application that serves as a centralised hub for information. Instead of using a text-based approach, the application uses modern technology and complex algorithms to facilitate user compliance. The target audience is represented by a predominant part of the population, specifically individuals who drive personal vehicles in countries that embrace this kind of frameworks. The goal of creating this application is to eliminate all barriers of entry, such as subscription fees, prior expertise in the domain or the need to comprehend legal aspects. By doing so, the navigation process becomes much easier and helps the audience to make informed decisions when buying a new vehicle, while also expanding the impact and reach of environmental areas.

The platform allows users to save vehicles to their account and make use of route planning features. The intuitive design allows users to quickly obtain the information they require, without going through extensive pages of content. Over more, the application features informative text-based pages, for comprehensive insights.

At the core, the application makes use of a robust database, manipulated by a set of algorithms. They process user input in order to determine the appropriate registrations for a vehicle and validate its access in LEZs. These technical aspects ensure the accuracy, reliability, timeliness and relevance of the provided information. Even though the application is designed for individual consumers, the database model and it's free-to-use nature make it suitable for enterprises, accommodating a larger number of scenarios, vehicles and routes within the same account.

Despite the focus set on simplicity and accessibility, the application keeps the same high standards for information quality, meaning it is up-to-date and accurate. In addition, it contributes to spreading awareness about environmental zones and promoting the social acceptance of urban vehicle regulations. Users are encouraged to take smart and responsible decisions, therefore boosting the effectiveness of the regulations imposed. Through its unique manner, the platform ultimately contributes to the larger scope of improving air quality in European cities.

The interface must be centred around the user, making sure to reduce the time required to obtain the information for his specific situation. While the software application does feature text-based pages for deep insights into LEZs, the algorithms used in the validation process stand out the most. They are encompassed by an intuitive interface that allows users to swiftly access the information they need. The actual process of triggering and using the algorithms is seamless and requires no technical background.

5. Application Development

5.1. Analysing System Requirements

The development of this web application was structured in different stages, in accordance with the principles of software application analysis and design principles highlighted by the University course [40]. The first step of the application's lifecycle development is to define the project theme. This means to determine the main objective and functionalities based on global requirements and constrains. Researching other projects, can also determine aspects that could be improved.

Already existing resources are text based and not accessible to everyone, either because the information is not concise enough and outdated, or because of the use-cost. In this case, the application must be easily accessible to the public and must integrate modern technologies to facilitate information access. Through its interface, the system needs to permit user account creation and registration of personal vehicles. The last one is beneficial when the user wants to verify if he is allowed to drive in a specific city that may include environmental zones.

The main feature is represented by the *Route Planner*, which includes access verification for environmental zones. This page must feature a dynamic map and provide access to information instantly. The information includes a statement result and other key details, facilitating user compliance to the regulations. This way, users do not need to personally research all the regulatory requirements when planning a trip, they only need to fill in all the interest points and select one of their previously saved vehicles. Additionally, the application offers the possibility to verify which registration can be assigned to the user's vehicles. Since not all European countries necessitate

registrations for accessing the environmental zones, this page will provide general information about the framework.

Therefore, the overall design of the software application can proceed. This step defines a high-level view of the system architecture, that includes the interaction between its components and a minimalist database model. The software architecture also needs to ensure communication between the user and the system, while the database needs to comprise information about users and environmental zones. Based on this data, the system uses special algorithms that determine whether a specific vehicle is permitted to access points of interest. Additionally, since there are many different registrations, each with different assignment criteria, the application must offer a way for users to determine which ones can be assigned to their vehicle. Notable constraints are represented by vehicle category and environmental zone type. In this case, the application focuses on passenger cars and Low Emission Zones.

The detailed design is built upon the specifications defined previously. So far, only the basic structure of the application has been created. Onwards, the particular details of the requirements are described. This includes the selected system architecture and technologies, as well as the extended database, which contains European countries and cities that have LEZs. The Client-Server model represents an obvious choice for the architecture, being perfectly suitable for this web application. The content of the web pages must fit the functionalities of the algorithms which determine vehicle access and eligible registration.

Further, the Client-Server model splits the system in two layers: backend and frontend. The main stages of the backend development are the configuration of the URL routes, the development of the algorithms and the creation of the database model. The frontend elements of the website were created after the backend algorithms and APIs were already functional at a basic level. By taking this bottom-up approach, focus on application functionality is prioritised. Consequently, the user interface is designed to fit the features of the backend layer.

5.1.1. Information Sources

The research process preceding the design and development of this application includes different types of resources and references, all of which are mentioned in the Bibliography chapter. Considering the fact that the scope of the application covers the relatively new and innovative topic of Low Emission Zones, traditional books are generally rare or non-existent. Nevertheless, online resources offer legitimate repositories and contain valuable information.

Gathering information on Low Emission Zones was primarily focused on governmental websites, such as European Parliament official websites under the ".eu" domain, official websites for National frameworks and other authoritative institutions. This ensures that the application provides the newest and most accurate information. While other online resources exist, the official ones generally provide the best explanation of the framework restrictions and requirements, compliance procedures, affected vehicles, while also clarifying the contradictions created by segmented information from other sources.

On the other hand, academic research or conference papers often include information on the newly emerged topics. While momentarily the focus of this application is not to analyse and depict the most efficient frameworks for environmental zones, an overall understanding of the topic is required to model the data. Other studies may reflect possible futures of this topic, for example the study by Statista.com [11], which highlights the abundance of Low Emissions Zones.

As part of the analysis process, reviewing existing applications provides practical insights. From a research perspective, it highlights how the information is presented. Unfortunately,

comparing the information from different websites often leads to contradictory results. In order to solve this issue, as mentioned in the second paragraph of this chapter, the research filters the information to include only official websites that are up to date. From a technical point of view, challenges faced by these applications reflect aspects that can be improved in this new project.

The frameworks, libraries and APIs used, such as Flask or Google API, were inspired by the official documentation page and software engineering books. These represent the perfect resources, as the developers have access to the latest updates, code examples, features compatibility and work patterns, but also frequent bug fixes or answers for FAQ (frequently asked questions). The database structure was modelled based on the principles and practices of relational database, described in multiple documentations and development tools. The support of software engineering community generally offers reasonable solutions to frequent problems, innovations or refinement ideas. But these are usually more exposed to bugs and necessitate full understanding of the software system in order to be used.

5.1.2. Application Use Cases

Once the project requirements had been specified, the functionalities of the system must also be properly defined. They represent the tasks fulfilled by the system and how it interacts with users. This is effectively structured and can be visualized using the UML Use Case Diagram.

In UML, the use cases are represented by an oval figure and are named suggestively. They represent possible communication scenarios between the system and external users. As part of the diagram, these external users are represented by a human figure and they are referenced as actors. This interaction between users and use cases can only be of association type, represented through a flat line. Use cases can be related through inclusion, meaning that the execution of one use case implies the execution of another case, or through extension, meaning that the use case is conditionally triggered by another [24].

The interaction between the users and the web system described in this document, was represented below in the Use Case Diagram, *Figure 1*. This diagram was created in the early stages of the system analysis, in order to define the functionalities of the system and how it should interact with the end-user.

As a short explanation of the diagram, the actor from the upper side represents regular users, who can initially access the *Home* page and those designated for national LEZ schemas. On these pages they can find general information about the application and how it works, but also text-based information about different Low Emission Zones. In order to gain full access to the features of the application, users must create accounts and save their vehicles. This is represented by the inclusion relationship, as in order to login, users must create an account. The same logic applies to the other inclusion relationships. Vehicle registration is also offered only for users with accounts created and logged in. Users have the possibility to edit the specifications of their saved vehicles.

The second actor in this diagram, from the bottom, represents the administrator of the application, who has a special account type which allows him to edit the database directly from the web application. The admin user, also referred as technical user, can also access the other functions of the application, as he is a generalisation of the user actor.

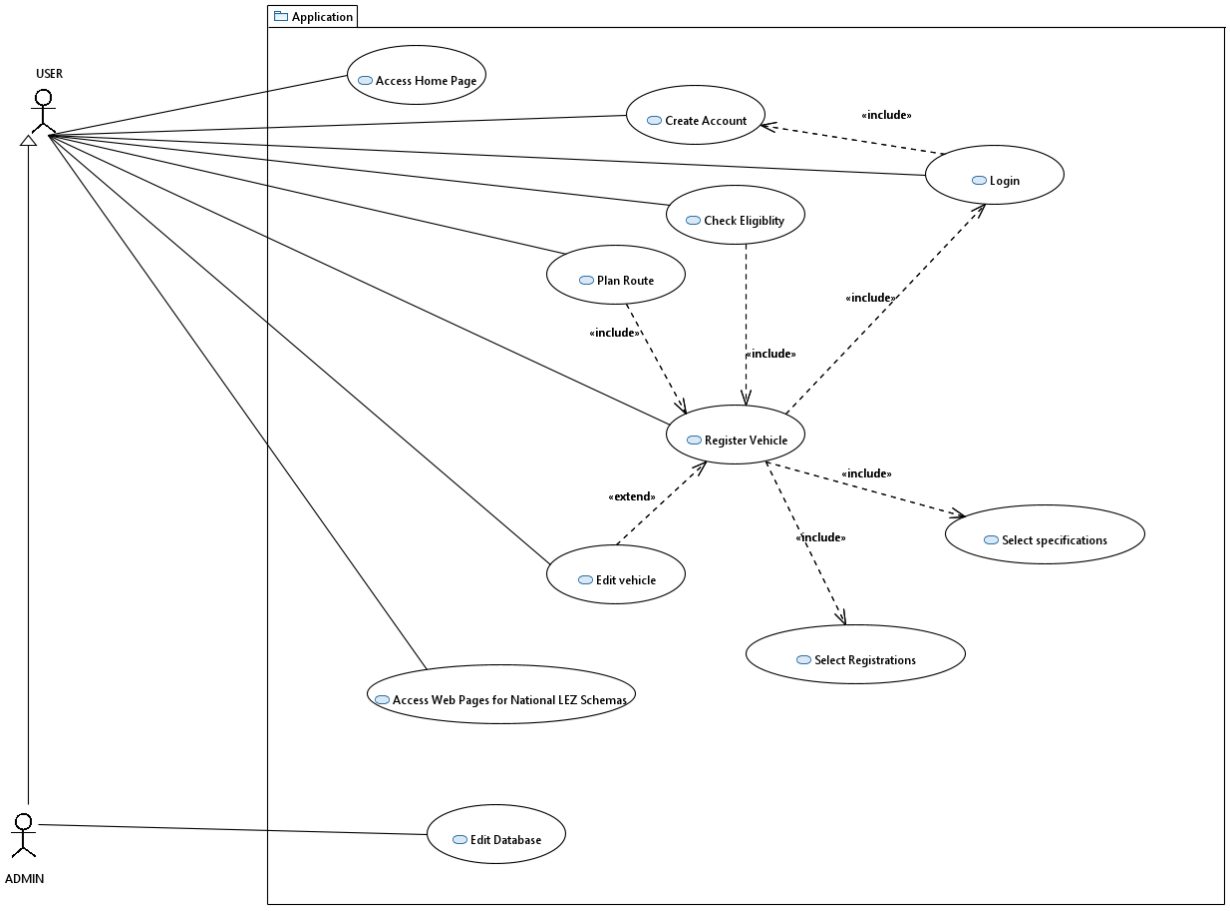


Figure 1. Use Case Diagram

5.1.3. Access Validation Process

The activity diagram is also part of the UML standard and represents the controlled flow of data within the system. This model is used to represent the dynamic aspects of a system and generally illustrates the sequence of activities triggered automatically. Activities represent the execution steps of an algorithm and only have entry and exit transitions associated. These transitions are generally called flow controls, represented by arrows and triggered automatically by the end of an activity. Additionally, in the activity diagram, engineers use specific elements to display conditional statements. The conditional node is graphically represented by a diamond-shaped figure, with one input and multiple output transitions, each representing a different condition. Another way to represent complex features in the diagram is to use concurrency elements when representing synchronization of parallel activities [24].

Figure 2 represents the activity diagram of the *Route Planner* page and depicts the procedural sequence triggered by the user. After the form is submitted via the HTTP POST request, the diagram depicts the asynchronous aspect of this event sequence, as the frontend interface interrupts the flow and waits for backend response. Afterwards, the notifications are displayed, which means the user has received all the information he wanted and the sequence is finished.

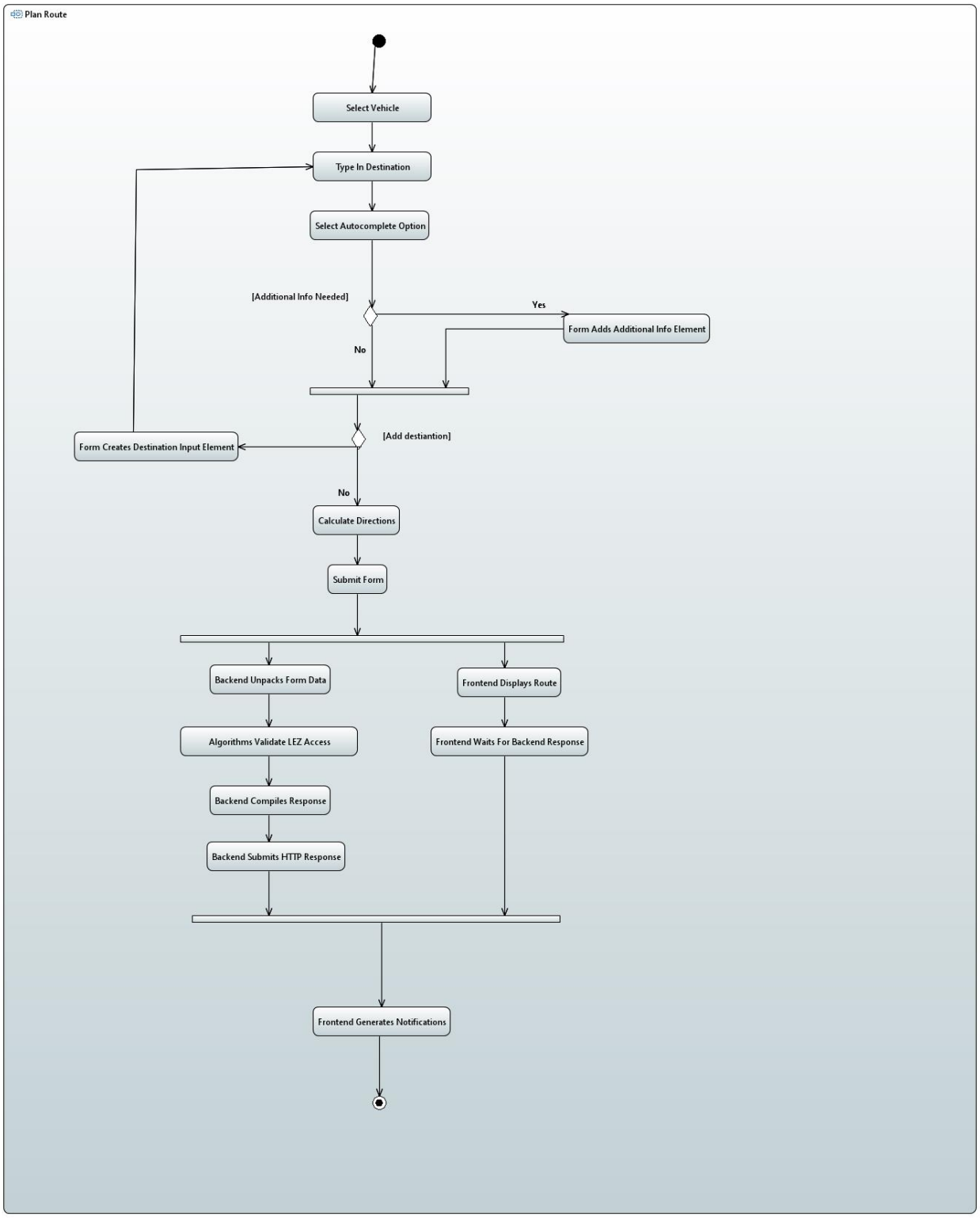


Figure 2. Activity Diagram for Route Planner page

5.1.4. Nonfunctional Requirements

The functional requirements refer to the capabilities and features of the application and its interaction with the users. Shortly, they can be referred as utility aspects of the system and were described above.

Nonfunctional requirements refer to application performance aspects, which in some cases may not be directly visible to the user but are essential in the functionality and stability of the system as they improve end-user experience [26]. The nonfunctional requirements of the application were defined in the system analysis process.

System scalability is necessary in order to cover future traffic regulations and vehicle types, while also maintaining stability during heavy user traffic. The database model of the application must easily adopt to such changes. Scalability also includes easy maintenance of all elements, including the adjustments needed in validation algorithms.

In order to ensure the application is reachable to a wide audience, the application must be available across different platforms, devices or operating systems. While web applications are generally supported on different devices, the user interface should be optimised to fit different screen sizes and work on different operation systems. Real time response to user input should be ensured by the system and maintain the performance even during high load. Security and integrity of the data is important. Therefore, user information must be encrypted and the system should be protected against unauthorized access, in order to avoid data corruption.

5.1.5. Application Constraints

The initial stages of this application have been developed to cover a specific segment of the market, therefore, its utility is limited to the requirements of those individuals. Nevertheless, the analysis of urban regulations and the development of the web application serve as fundamental steps in creating the central hub for traffic restrictions and regulations.

Firstly, the application is limited to a single class of vehicles, specifically passenger cars. The choice to cover these vehicles was made because they represent one of the most widely used personal transport methods and are affected by the newly European regulations.

Secondly, the web application momentarily only addresses Low Emission Zones. While environmental zones take different forms and impose different strategies for reducing pollution, LEZs are affecting the majority of population [1]. This results from the nature of the restrictions, as they target central areas in large cities. They are the most efficient form of pollution reduction in urban settlements. To create a data model for different restrictions and to use it in a universal access validation system is also a challenging aspect.

5.2. System architecture

The Client-Server model represents one of the most used architectures in system development. This model splits the actors that take part in the interaction in two: the Server, responsible for hosting and processing resources, and the Clients, which request the resources. Generally, the communication takes place via the internet and allows multiple users (Clients) to connect to the Server at the same time. As depicted in the Journal of Computer Engineering [41], the Client-Server architecture allows systems to easily collect user information and process the data in order to provide fast responses.

The web application presented in this document is designed following the Client-Server model. In this case, Clients are represented by users, specifically by their web browsers, which send HTTP requests. The Server is basically the web application, which is running on the local

machine and is started in the built-in Flask development mode. This is ideal for application testing, which takes place during the development phase. Binding the Flask server to the local IP address allows other devices to connect to the application and send requests.

When users navigate across the website and perform actions, the Client sends HTTP requests to the Server. The Flask backend of the application receives the requests and routes them to the appropriate handler functions. These functions are responsible for processing requests, whether it contains data that needs to be saved or refers to a new page that needs to be rendered. Additionally, the Server communicates with the database to store and retrieve information, as dictated by the Client's request.

This architecture ensures that the application is scalable, maintainable and improves efficiency by centralizing processing tasks on the Server. Thus, the Client-Server model used even in its current local deployment, effectively manages user interaction and suits the requirements of the application to create an efficient and user-friendly web application.

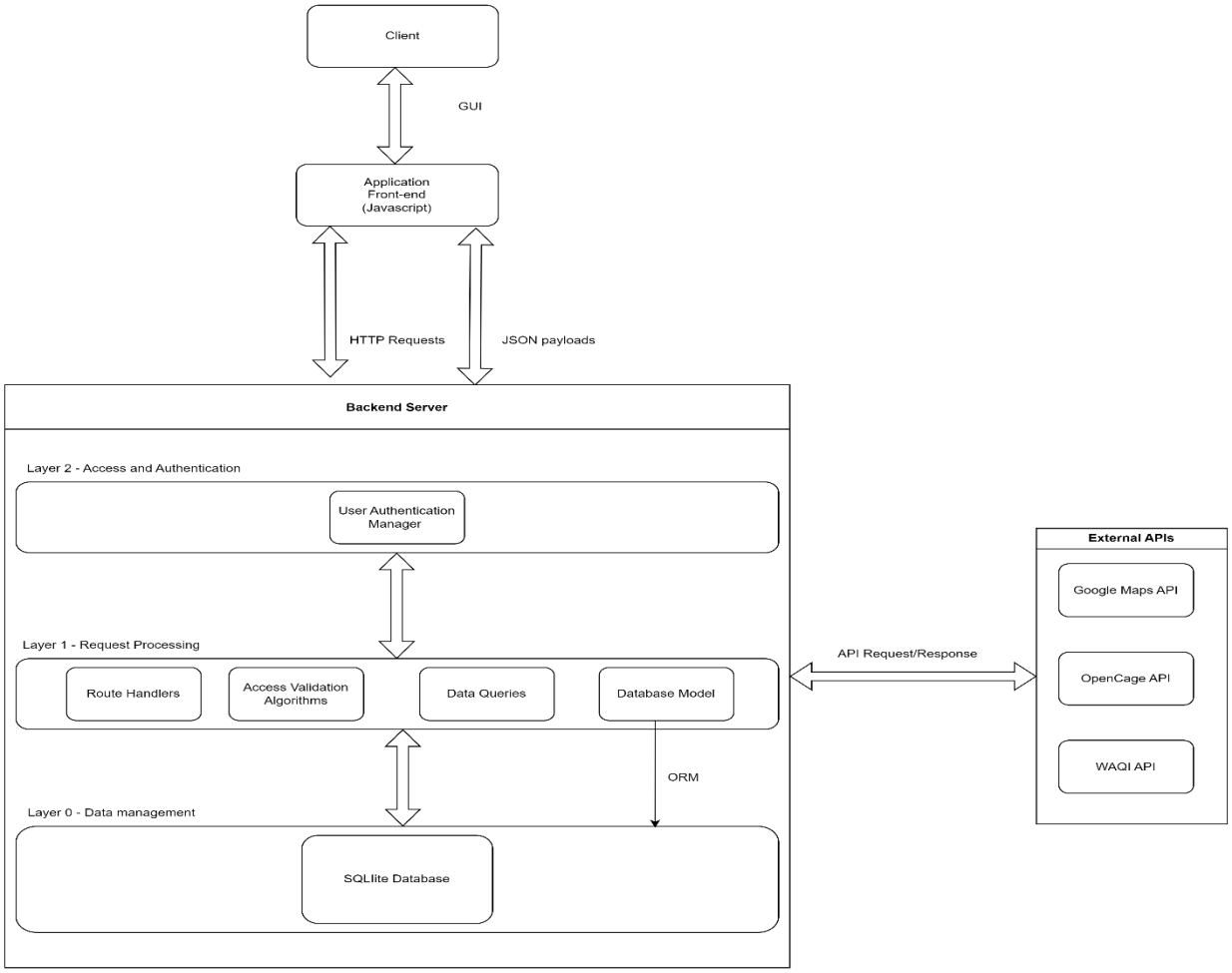


Figure 3. System Architecture

5.3. Backend Elements

5.3.1. Route Handlers for User Interaction

The route handlers, also referred as request handlers or endpoint functions, are algorithms written in programming code and define system functionality. These algorithms are triggered when the user accesses a specific URL or makes an HTTP request. Since the application is based on Flask, the route handlers are standard Python functions with specific decorators. These decorators take two main parameters. The first parameter is a string that reflects the URL or endpoint served by the function. The URL is the path accessed by the web browser to retrieve a specific resource, in this case, predominantly a HTML page. The second parameter of the decorator is called *methods.* It is a list of strings and refers to the types of HTTP requests corresponding to the route handler. These request types ensure the CRUD operations. Notably, one route can be responsible for more than one operation, but it does not need to fulfil all of them. As an example, when the user wants to register a new vehicle to his account, he accesses the *"/new-car"* URL endpoint. The Python function that is triggered by accessing the endpoint is preceded by two decorators. One of them is the Flask classic *@login_required* that checks the current state of the user, in order to validate his authorization [39]. In this case, this decorator is necessary because only users with an account logged in can register a new vehicle. The other decorator marks the function as handler for GET and POST requests of the specified endpoint.

The submission of the POST request is triggered when the user presses a specific button. The body of the request contains the information filled in the web form. The algorithm checks if the received request is of type POST before retrieving the data submitted by the user. The data is accessed using the Flask *request* object, specifically by the *form* attribute. In order to properly manipulate the data, *request.form* is formatted in a dictionary variable. Then, classic dictionary indexing (*request.form.get['item']*) is used to access the data. Using the retrieved information, the algorithms create a *Car* object and append it to the corresponding database table. This is an instance of the *Car* class, which had been previously defined in *models.py* and mapped to a database table by the Object Relational Mapper SQLAlchemy.

GET request is automatically submitted when the user accesses the URL. By this, he requests to retrieve the HTML page dedicated for registering a new vehicle. In case the user makes a GET request, the endpoint handler uses the *render_template()* function offered by Flask library. This operation is standard for Flask's templating engine and integrates dynamic data into HTML templates.

The *render_template()* function also has some arguments [39]. The first argument is mandatory and refers to the name of the HTML file requested by the user. Specifically, when registering a new car, the user will be directed to the designated page, which is *new-car.html*. This file is stored inside the *templates* directory, along with the rest of the pages that are part of the interface. Further, the *render_template()* accepts a variable number of arguments. Generally, these represent the state of the system or data that is sent to the frontend and can be used in the dynamic template rendering. In the context of this application, the *render_teamplate()* always takes the parameter *user=current_user*. As the name suggests, *current_user* is the user currently authenticated and interacting with the web application. By passing this object to the template, under the variable name *user*, the web page gains access to user-specific information, such as the name, saved vehicles or other database information linked to the primary key of the user. This facilitates the generation of personalized content, based on the available information about the

authenticated user. In other cases, the *render_temaplate()* also takes variables or data that had been previously calculated by the backend processing algorithms.

Another way this application ensures user interaction and handles HTTP requests is through REST APIs, also defined in the *views.py* file [39]. They handle the retrieval of resources with the GET request or update the resource using the POST request. Data is transferred between the client and the server (or reverse) using the JSON format. For each endpoint accessed by the user, the application triggers designated sets of algorithms that process the request. This method adheres to the RESTful methods and principles, handling each request separately and serving users with the desired information.

When clients use the application, the constant communication between the backend and frontend also implies quick data transfer. Python offers dedicated JSON library, which was imported in the application environment together with Flask's features to handle the creation, manipulation, retrieval and submission of data in JSON format. The frontend of the system uses JSON to store and submit user-input data via POST or GET requests. In the backend, the data is unpacked by the algorithms. After parsing the JSON elements, the algorithms can process the information to compose a response. This response may contain data which needs to be submitted back to the frontend, in order to be displayed to the user.

Summarily, this web application uses both Flask's standard routing and REST APIs. The first technique uses predefined functions, such as *render_template()* and *redirect()*, to load new HTML pages in the frontend. On the other hand, REST APIs are used when client action requires data processing without rendering a new HTML page. For example, to select the registration eligible for a vehicle, the system uses REST APIs to process user input and submit a JSON response to the frontend. Then, data is fetched and rendered in the same HTML page.

These mechanisms facilitate user interaction with the web application, improving to the functionality of the system. Using JSON format ensures the integrity of the data and compatibility with the two programming languages[35] (Python for backend and JavaScript for frontend). Also, because of its structure, the debugging process was fairly intuitive, as the plain text of this format is readable by humans.

5.3.2. Database Model

As mentioned before, it is typical for Flask applications to interact with the database using Object-Relational Mapping libraries. In this case, SQLAlchemy is used for the creation and management of the database. This library allows to define database tables using Python classes. In Flask applications, the classes are created and managed in the *models.py* file [39].

The CRUD operations are managed by the endpoint handler functions, which interact with the database using Python operations, such as *db.session.add()* or *query.filter_by()*. ORM does not restrict the usage of classic SQL syntax. Once the database file has been created, SQLite operations can still be executed.

The schema of the database was created by mapping the classes to actual database tables. Then, through DB Browser graphic interface, the tables were populated by executing SQL commands. Considering the complexity of the data model, this software was extremely helpful for data visualization, but also for modifications when necessary.

Considering the fact that this application was created in Flask, another aspect that needs to be mentioned in regard to the database management is the migration scripts. This feature is part of the Flask-Migrate extension and integrates SQLAlchemy. In essence, these migrations represent changes to the database schema, which only become effective after the commit action takes place.

Such modifications include adding or removing tables and modifying columns of already existing tables. Special scripts automatically generate the migrations, based on the latest modifications detected in the *models.py* file. Once generated, migration updates are applied in an incremental approach to the database schema. Flask-Migrate tracks the current state of the database schema and applies migrations to bring the database schema in sync with the defined models. One of the biggest advantages of migrations is the possibility of version control, as all versions are saved in the *migrations* directory [16][39].

Structurally, the *Zone* table is responsible for storing the data about the Low Emission Zones. This includes key information regarding vehicle access, such as required registrations or minimum Euro emission standards. This data is processed by backend algorithms and usually queried by class functions that are also defined in the *models.py* file.

ZoneTemporaryData table is related to the *Zone* table through a foreign key. Specifically, each instance from *ZoneTemporaryData* belongs to a single instance in *Zone* table. As its name suggests, *ZoneTemporaryData* table is designed to hold data regarding the temporary characteristics of some LEZs. This is especially relevant for countries such as Italy or Bulgaria, which have Winter Low Emission Zone schemas. Such restrictions are only effective during a specific time of the year. Outside this period, different or no restrictions may be imposed, therefore the algorithms need to have access to both scenarios in order to validate vehicle access. In the class creation code, the *backref* parameter creates a back reference in the *Zone* model. This allows accessing *ZoneTemporaryData* instances through their corresponding *Zone* instance. The *temporary data* attribute was defined to facilitate data fetching when using just Zone instances.

Another essential part of the database model is the *User* table and the ones associated to it. The *User* table is created to store the essential data for account creation. The amount of information is minimal, as the nature of the application does not require much. This is also favourable for safety and resource storage reasons. The main fields, email and password, are needed for user login. In order to keep the same standards as most modern applications, the database stores a hashed version of the password. The encryption was made by using the SHA256 algorithm, which is one of the most secure and popular hashing algorithms [42]. This feature is provided in Python, by importing the *generate_password_hash()* and *check_password_hash()* functions from *werkzeug.security* library [16]. For the login and registration process, the system defines separate endpoints.

Creating a personal account for each user is important, however this would be pointless without the possibility of storing personal data related to this account. In this application, the user has the possibility to register his vehicles and save routes.

As both names suggest, *Car* and *SavedRoute* tables store the relevant data for the user's account. They are related to the *User* table through a foreign key, which creates a one-to-many relationship. This means that a user can have multiple *Car* instances or saved routes associated to his id. While in reality a vehicle has many specifications, the application is designed to store only the data relevant to validating LEZ access.

Routes are saved in a separate table (*SavedRoute*), which is also related to the *User* table through the foreign key. Once again, the one-to-many relationship implies that a user is able to save multiple routes. This comes as a big advantage for frequent travellers and enterprises. As for now, the maximum number of destinations in a route is not limited. The destinations are saved in JSON format and stored in *destinations_json* field, used in the software logic. However, they are also saved in *destinations_text* field, which is used to store the human readable version of the data.

Over more, each country that has a Low Emission Zone (therefore a table associated to a Python class) extends the abstract class *GeneralRegistrations.* This means that for each country, a

table is created to store the available registrations or LEZ entry criteria. The object-oriented principles allow the creation of the abstract class, serving as the superclass (parent class) in this case. It can be extended by other classes, referred as children or subclasses. The inheritance properties allow the subclasses to use or redefine the parameters and the methods from the superclass. Since *GeneralRegistration* is an abstract class and serves as a blueprint for the tables storing country specific registrations, it is not mapped to an actual table in the database. The superclass defines the relevant attributes and implements the method *find_best_registration_badge()*. This is used in the *Eligibility Check* web page. The rest of the classes extend the *GeneralRegistation* and some of them redefine the method or add some additional attributes. For example, *PolandRegistrations* class creates the columns necessary for storing the minimum criteria based on Euro standard and on the vehicle's last registration date. This was necessary because of the ongoing regulations in Polish Low Emission Zones. Obviously, the method of selecting the registration differs from the standard one defined in the superclass.

Figure 4 represents the relational database model of this application. This representation was made using the UML class diagram. The building blocks are classes, mapped to tables in the SQLite database. They are connected by lines, representing the relationship between them [24].

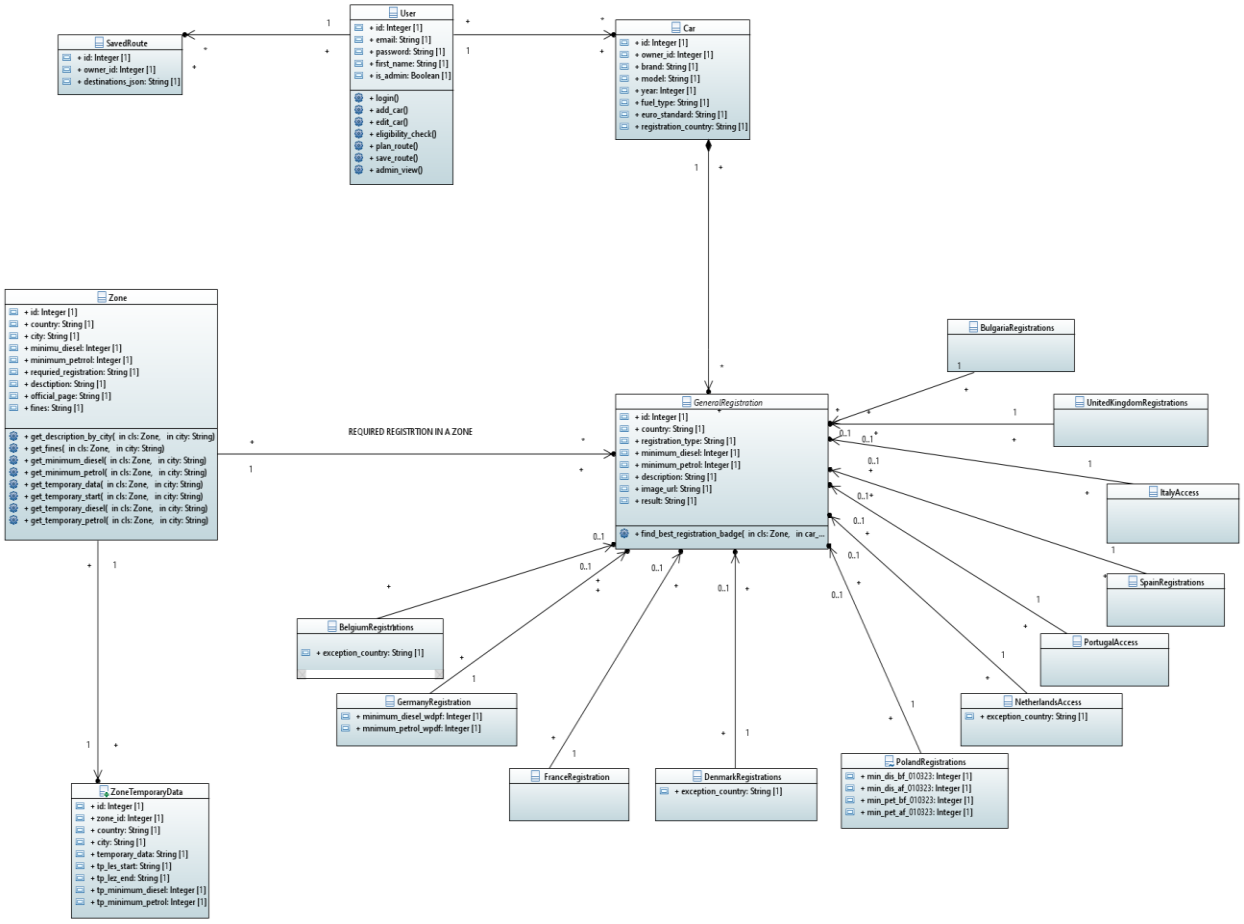


Figure 4. Database model

5.3.3. Route Planner and Access Validation

This system encompasses a different algorithm for each country that has a LEZ schema. Selecting this strategy is necessary for ensuring the validation of all possible scenarios. System scalability is mandatory due to the evolving nature of the Low Emission Zones and this structure guarantees easy implementation of future upgrades. In order for the application to stay relevant when regulations change, slight adjustments might be needed in the access validation algorithms.

Separating the algorithms for each country means the backend can be viewed as a set of interchangeable modules. Each module represents a country that has a Low Emission Zone schema and is structured in three parts: the validation algorithm, registrations/minimum requirements for access and the country instances stored in the *Zone* table. Since each national framework evolves at its own pace, a modular approach is needed for an easier maintenance. This is beneficial for both the developer and the stakeholders, as deploying updates becomes much simpler. While one task is to research and monitor the evolution of LEZs, the developer can implement small updates only in the specific module. This software architecture ensures minimum downtime of the application and quick bug fixing.

The validation algorithms are used on the *Route Planner* page to verify vehicle access in LEZs and display a short summary of the relevant information. When the user presses the associated button, the POST request is sent to the application backend and the relevant algorithms are triggered. Specifically, only the algorithms corresponding to the countries in the request are triggered. Since most countries have different restrictions from one region to another, the algorithms are designed to check for the city received in the request. Running a separate algorithm for each country proves to be effective in handling all the possible scenarios (LEZ access granted or forbidden, LEZ active/inactive during selected travel period, city or country does not have LEZ, etc.). This is especially efficient in processing requests for several points of interest at the same time. Additionally, saving results from multiple algorithms is extremely efficient with JSON.

While the frontend interface of the *Route Planner* page seems simple, the database queries, the POST request and the validation algorithms are all processed in the backend. Since one request can include several points of interest, they are stored in JSON format and loaded in a list variable. As the algorithm parses the list, it calls the corresponding access validation algorithm for each entry. These access validation algorithms decide the type of notification, success or error, and the details that will be displayed in the frontend. The *type* attribute is relevant in creating a suggestive design of the notification. The text of the notifications is queried from the *Zone* table through a series of class methods. These methods are created inside *models.py* file and take the city name as a parameter. Each of these methods retrieves specific information for the validation scenario. This includes short descriptions of the LEZs, penalties for not complying, official authority's webpage etc. All this data is then sent as a JSON response to the frontend of the application. There, the JavaScript layer of the system fetches the data. The instant communication between the frontend and the backend grants a smooth experience for the user-friendly page. This process abstracts and encapsulates the complex logic of the application and hides it from the user, who is only interacting with the system through the graphic interface.

5.3.4. Eligibility Check

This feature can be accessed from a separate web page, therefore it has a specific URL. In fact, the backend includes two dedicated route handlers for this endpoint. This happens because the user has two possibilities of checking which registration is viable for his vehicle: select a

vehicle saved to his account or introduce the specifications of a new vehicle. Different endpoint functions are triggered depending on the user's choice.

In the first case the route handler fetches the vehicle data from the database, while in the second, the request contains vehicle's specifications. In both cases, the request also contains the country for which the user wants to test. Furter, both handlers trigger the correct algorithm (to select the registration for the specified country). They build a response, fetching the database to get the details of the correct registration. The two route handlers are REST APIs and submit a JSON response to the frontend.

5.3.5. Dynamic Rendered Web Pages

Creative software system architectures also take advantage of the database when rendering web elements. This removes the need to manually create all the elements of the HTML files. Specifically, the Low Emission Zones data, stored in *Zone* table, dynamically renders part of the web pages with information about each country. The URLs assigned to these pages are dynamically generated when the client requests to access one of them. In the backend, the route handler takes a parameter which represents the country, extracted from this URL. Based on this attribute, the route handler makes database queries, processes the data and structures the response. The generated HTML file is rendered with *render_template()* function and takes parameters containing the data processed in the route handler. One of the dynamic elements created in the DOM of these information pages is the section that contains the LEZs. Basically, this is an unordered list, created by the *<ul>* tag and populated by the elements received from the backend. This method not only highlights the capabilities of the Flask templating engine, but also reduces the repetitive code considerably. In classic HTML style, in order to achieve the same result, each list would need to be created manually. This would be redundant, as there are several countries and the Low Emission Zones are already saved in the database. Moreover, by using dynamic rendering, it is enough to implement the updates in the database, without additional changes to the HTML files.

When using *render_template()* in order to display the HTML page, the files are rendered on the server side, meaning the HTML files are fully generated before being sent to the client [16]. For some web pages, the application combines template rendering with REST APIs for the CRUD operations. In contrast, if RESTful methods are used, the client consumes the API and based on the received data, it updates or generates the user interface. While the effects of the dynamic templates are visible in the frontend interface, their implementation was created in the backend.

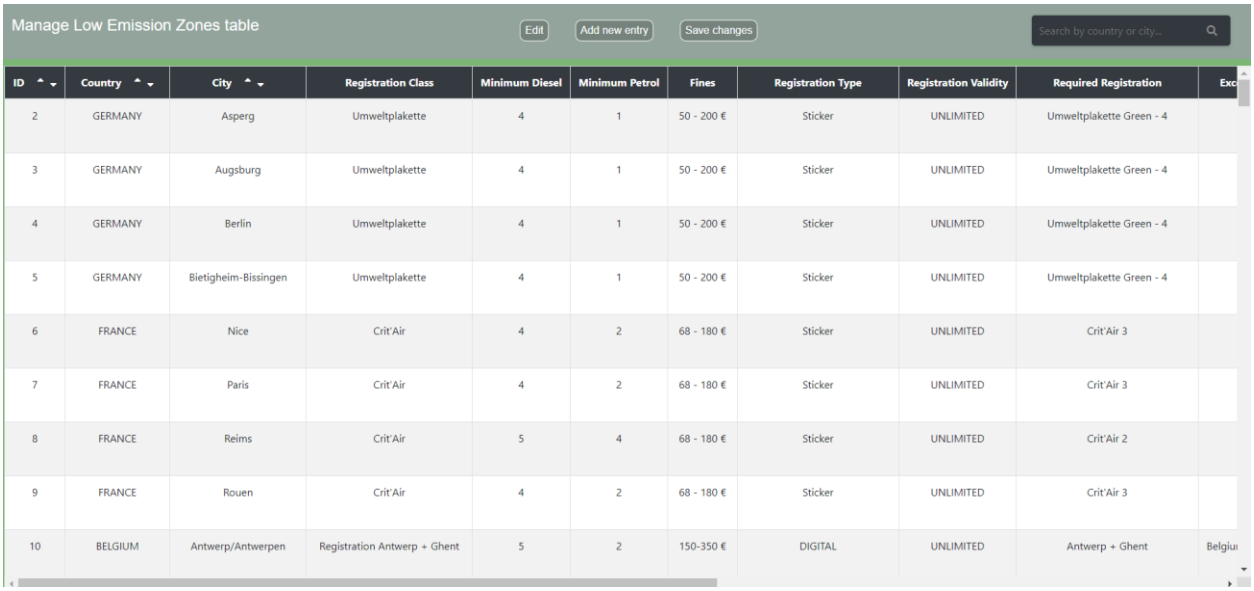
5.3.6. User Types

As explained previously, the database stores SHA256 hashed versions of the passwords, in order to ensure the integrity and security of the application and its users. Flask provides a handful of tools for managing client authentication process. User sign-up, login and logout, all have dedicated request handlers, defined in *auth.py* file.

Considering the nature of the application, the authorization process takes into account three types of users. The first category is represented by people that do not have an account, possibly even first timers. Their access is limited by the *@login_required* decorator associated to the route handler functions.

Secondly, there are regular users, who already have an account and are signed in in order to gain access to the rest of the features. Each user session is handled by the application backend and authorizations are verified by the *current_user* object offered by Flask [39].

Lastly, the application authorizes access to a special admin user, which represents the third category. This one has been created with the purpose of manipulating the database, directly from the dedicated application page. The admin user role can be assigned to stakeholders, even to those with little technical background, as they benefit from a user-friendly page, which makes it easier to edit the Low Emission Zones table. These users are able to make database modifications, when necessary, without the involvement of the developer. The administrator account, also referred as technical user, is essential in modern applications, as stakeholders should have the possibility to manage the database. However, for security and data integrity reasons, this role must not be assigned to any person. Accessing the endpoint for the admin dashboard is only possible for users who have this authorization set as true in the database. The *User* table has a specific Boolean field that is checked by the algorithms before rendering the admin dashboard.



ID	Country	City	Registration Class	Minimum Diesel	Minimum Petrol	Fines	Registration Type	Registration Validity	Required Registration	Exc
2	GERMANY	Asperg	Umweltplakette	4	1	50 - 200 €	Sticker	UNLIMITED	Umweltplakette Green - 4	
3	GERMANY	Augsburg	Umweltplakette	4	1	50 - 200 €	Sticker	UNLIMITED	Umweltplakette Green - 4	
4	GERMANY	Berlin	Umweltplakette	4	1	50 - 200 €	Sticker	UNLIMITED	Umweltplakette Green - 4	
5	GERMANY	Bietigheim-Bissingen	Umweltplakette	4	1	50 - 200 €	Sticker	UNLIMITED	Umweltplakette Green - 4	
6	FRANCE	Nice	Crit'Air	4	2	68 - 180 €	Sticker	UNLIMITED	Crit'Air 3	
7	FRANCE	Paris	Crit'Air	4	2	68 - 180 €	Sticker	UNLIMITED	Crit'Air 3	
8	FRANCE	Reims	Crit'Air	5	4	68 - 180 €	Sticker	UNLIMITED	Crit'Air 2	
9	FRANCE	Rouen	Crit'Air	4	2	68 - 180 €	Sticker	UNLIMITED	Crit'Air 3	
10	BELGIUM	Antwerp/Antwerpen	Registration Antwerp + Ghent	5	2	150-350 €	DIGITAL	UNLIMITED	Antwerp + Ghent	Belgiu

Figure 5. Admin Dashboard

5.4. Frontend Elements

5.4.1. Jinja Templating

One of the advantages that comes with the use of Python, specifically Flask for backend, is the possibility to use the available templating engines [39]. While the aspect of rendering dynamic web pages with the pre-processed backend data was described in chapter *Dynamic Rendered Web Pages*, further frontend details must be presented on this topic. The application uses Jinja templating across all web pages, which reduces code redundancy and makes the creation of the DOM much more effective. In general, all the web pages of a web application have the same design theme, meaning they look similar because they often reuse some elements. More specifically, the web pages of this application are created as an extension of *base.html* file. In Jinja, this implementation is relatively simple, yet extremely effective. Creating the base template includes the *{%block%}* elements. Inside of these blocks, classic HTML elements are used to create the structure of the template. Then, in the rest of the files, the extension is made by using the key block *{%extends file.html%}*. This method allows effective reuse of elements across different pages.

In any web application, the navigation bar represents a key element, as it assures smooth navigation across the website. Conventionally, this element contains buttons for accessing the most important functionalities and is present on the top of all webpages. For this application, the navigation bar element (navbar) was created in the *base.html* file. This *base.html* is not accessible to the user because it is just a template. This restriction is imposed by the fact that none of the defined endpoints return the *base.html* file. Additionally, in Jinja templating, the CSS and Script files linked to the base template are also made available in the files that extend it. This means that the scripts for the actions or the styling of elements used on different pages can be defined in their specific stylesheet or script file and linked to the base file.

5.4.2. Dynamic Map Element

On the *Home* page, the available Low Emission Zones are displayed, with the help of Google Maps API. When the client accesses the specific URL, it sends a GET request to the server, which queries the data and integrates the LEZs in the response. JavaScript functions fetch the data from the JSON response and extract the countries and cities. In order to create a marker based on a location, the address must be put through the geocoding process. This process is completed through the API offered by OpenCage [29]. Then, the frontend creates the map element and places the corresponding markers.

Additionally, the maps displayed in the interface are customized by the specific attributes offered by Maps API. Slight adjustments to the default behaviour of the map element allow users to roam freely [28]. Meaning, users can manually scroll and zoom to explore the map in order to identify the different LEZs. Mouse hovering over a LEZ marker displays the name of the city and real time air quality index. This data is gathered using the WAQI API [30].

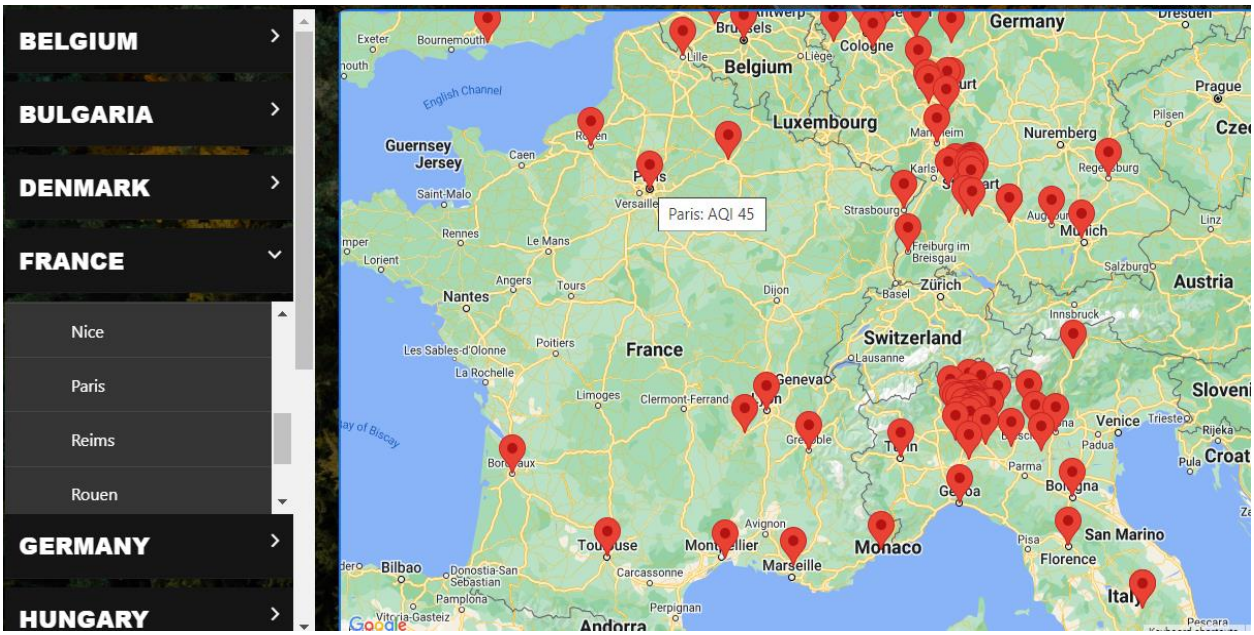


Figure 6. LEZ Map displayed on the Home Page

5.4.3. Navigation and Directions API

Further, since the main highlight of the application is the *Route Planner* page, this feature would not be complete without the navigation elements. The user is guided to fill in the form with his destinations. Afterwards, the map element displays the best route between the selected points. This route is calculated by the Directions API from google, which offers the best results [28]. The map element is updated in real time, without the need to open a new session or refresh the page. This is achieved by determining the directions through a JavaScript function, using the navigation feature of the Google API. This JavaScript function is invoked by the asynchronous function designed to handle the form submission. When the user clicks the *GO* button, the web browser submits a POST request containing the data from the HTML form. From that point, the asynchronous function waits for the response in order to update the map element. As expected from any navigation, the map displays markers for the selected locations. The geocoding process is again necessary, as the user introduces an address, not geographical coordinates. This time, the process is done using the Google Geocoding API, not through an additional third party.

The choice of splitting the geocoding process between two service providers was made for two reasons. One of them is the fact that using APIs from big providers comes with an associated cost. In order to limit the traffic generated by the Google Maps API, the API key used in this application has set usage limits. Since the costs associated with OpenCage are much lower than those of Google Geocoding API, the *Home* page uses OpenCage for the geocoding. Here, for any load, the application makes a few hundred requests, directly correlated with the number of LEZ cities stored the database. On the other hand, the *Route Planner*, in most cases should generate a much smaller number of API requests, therefore Google Geocoding API is a viable solution. The second motive is that integrating different technologies in the beginning stages of a project opens up multiple scenarios for future expansions.

The coordinates obtained from geocoding are necessary for the Maps API to create and display markers precisely. Since the database only includes fields for countries and cities, the geocoding process reduces the complexity of the table storing LEZ, but also simplifies the process of adding new entries. The alternative would be to add a field in the *Zone* table that stores the coordinates for each city. Even so, geocoding would still be necessary in the *Route Planner* page, as the possibility of geographical coordinates and addresses is infinite. The user can select not only cities, but also full addresses.

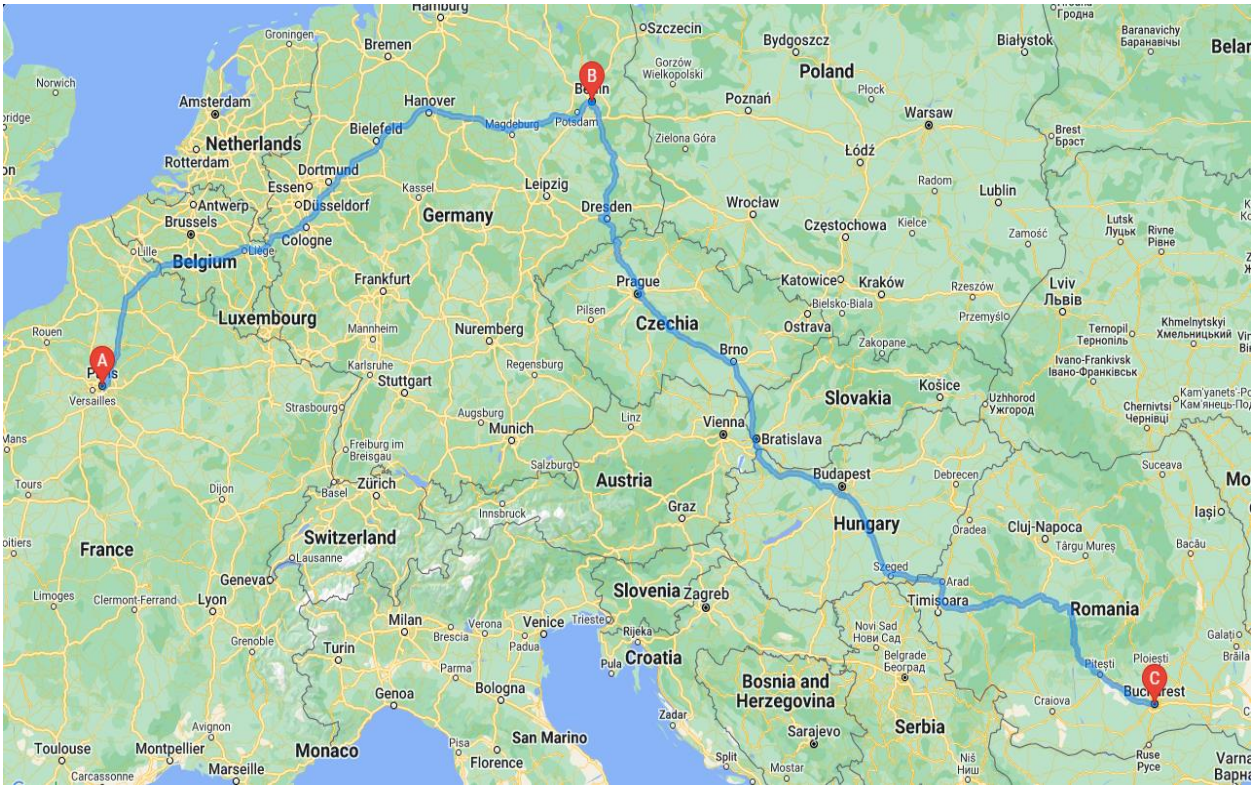


Figure 7. Route Planner

5.4.4. Asynchronous communication

The system enables an asynchronous communication between the client and the server on the *Route Planner* page. By pressing the *Calculate Directions* button, the user triggers an asynchronous function. The frontend layer gathers the user-filled information and formats the data in a *FormData* type variable. This object maps data in a dictionary-like format, associating key and value pairs which are ready to be sent via the HTTP request. One of the keys in *FormData* object refers to the destination address that had been previously saved in a JSON variable. The *FormData* object is included in the POST request that is be sent to the server. Afterwards, the frontend interface suspends execution until a response from the backend is received.

This synchronisation between the frontend and the backend was depicted above, in *Figure 2*. When this diagram was designed, the technological aspects were not clear yet. However, it was sure that the communication between server and client needs to be smooth. JavaScript asynchronous functions are the proper solution for this.

Basically, the process is paused (using the *await* key word) [43]. Therefore, it creates time for the backend layer to receive the data, process it and send the response. Frontend interface continues the workflow after the backend algorithms run and formulate the response.

This creates a seamless interaction between the user and the system, as generating the response is really quick. This technique allows DOM manipulation and real-time updates of the interface without the need to refresh. While the communication is started in the frontend layer, this functionality is also made possible by the rigorous endpoint functions defined in the backend.

5.4.5. Notification display

These notifications are designed to contain essential information and display it as fast and concise as possible. While the access validation algorithms are part of the backend, they are tightly connected to the frontend as well. The notifications described in these paragraphs are the result of the access validation algorithms. They interact with the database to determine the text and the type of the notifications. Based on this text, the notifications are created dynamically by the JavaScript algorithms in the frontend layer. Essentially, they create <*div*> elements which are then appended to the HTML page.

JavaScript algorithms set the class and id attributes of the alert elements according to the notification type. These attributes are pre-configured in the stylesheets to suggestively design the notifications. For example, if the access validation algorithms decide that the selected vehicle is not eligible for access in a LEZ, a red alert is raised. Color-coding the notifications is a common approach as red or green colours already have a signification in the population's mind. By appending pre-defined elements to the DOM, the frontend layer is ready to render the elements correctly. Since the number of notifications is dictated by the number of destinations, the flex-box approach is used to automatically fit all of them on the screen.

All notifications follow the same structure. First, a summary sentence states the result of the verification, whether the vehicle is eligible or not to access the LEZ. Then, a short description of the LEZ and the required registration or minimum euro standard, link for accessing the country page for further information and link for accessing the official authority website are displayed.

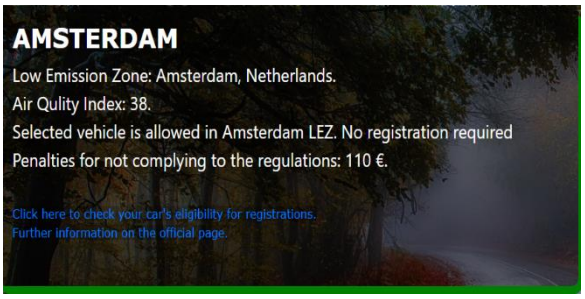


Figure 8. Access Granted Notification

Figure 9. Access Forbidden Notification

5.4.6. Dynamic Elements

This section describes how user interaction affects the structure of the web pages. This refers to data or HTML elements that are displayed conditionally, based on user input. For instance, in the webpages designated for adding or editing a vehicle, the application presents a dynamic HTML form. Rather than asking users to complete a long list of items with all the existing registrations, which would result in numerous blank fields, the interface guides them to only select attributes relevant to their specific vehicle. The selective retrieval of data also improves the efficiency of the system, reducing both processing time and resource utilization. These dynamic fields are accompanied by remove buttons to facilitate easy error correction and future updates.

For new users, with minimal knowledge of Low Emission Zones, the multitude of registrations might seem overwhelming. Therefore, they are initially guided to select a specific country before proceeding to choose the actual registrations. For example, a user from Germany

may be unfamiliar with the term "Distintivo Ambiental" or “Crit’Air”. This way, the process of selecting the appropriate registration for their vehicle becomes more intuitive.

Additionally, the notifications showcased in the previous chapter are also dynamic HTML elements. Their presence is dictated only by user action. As outlined, the information and the type of these notifications are dynamically generated. Furthermore, when selecting destinations in some specific countries, such as Italy, users are prompted to select the period of the trip. This is very significant, particularly because of the different LEZs regulations during the winter period, and is taken into account by the validation algorithms. However, for most countries, this requirement does not apply and, therefore, soliciting it would be meaningless.

To achieve the described behaviour, the application uses custom JavaScript functions and event-listeners that add and remove HTML elements. These functions utilize the Document Object Model to create new elements, set their attributes and append them to the HTML file [22]. Additionally, there are similar functions for removing the elements. These traverse the DOM to locate and delete the targeted elements, ensuring efficient management of dynamic content on the page.

5.4.7. Conditional Selection

While granting users the freedom to navigate the application is favourable, guidance in certain situations prevents errors. This precaution ensures that the validation algorithms receive accurate input. Given the wide variety of Low Emission Zones, the access validation algorithms cover multiple scenarios, therefore it is mandatory for them to receive accurate input. In addition to users receiving correct responses, which aids to regulatory compliance and avoiding potential fines, guidance through certain selections may also enhance overall experience.

One example of such situation would be the process of registering a new vehicle. In this case, users are prompted to input the brand and model for identification purposes. Subsequently, after selecting the brand, the model field is automatically populated with relevant options corresponding to the chosen brand. While technically this does not improve the system functionality, it improves the user experience, which is an essential requirement. Similarly, users will select the Euro standard based on their fuel type. For example, choosing the zero-emission Euro standard is possible only if the vehicle was previously marked as electric. Consequently, an electric vehicle can only be registered as zero-emission. While users are responsible for verifying the specifications in the vehicle’s registration documents, these restrictions prevent potential errors.

Another notable aspect of how the interface guides user interaction is present on the *Route Planner* page, when users select points of interest and benefit from a well-known autocomplete functionality. The suggestion options are generated by Google Places API. This API stands out as one of the most widely used autocomplete tool, being specifically designed and configured for location prediction. This software is extremely versatile, as it includes various points of interest, from city selection (usually reflects city centre) to full addresses that include street names or even numbers, hotels and cultural monuments.

Moreover, the use of autocomplete function ensures accuracy of selected places, guaranteeing precise input for the Google Geocoding API and the access validation algorithms. In order for the path and the markers to be displayed on the map, the geocoding process must convert the locations into geographical coordinates. Any small mistake in the address could fault the geocoding input data, resulting in inaccurate or missing results. Also, the autocomplete API offers full information on any selected place. For instance, when selecting a hotel, the autocomplete

feature automatically includes the country and city, which are then extracted by the geocoder for proper submission to the backend. As described previously, the city and country are the input for the access validation process.

In summary, these features are not implemented to block user access. Rather, they are designed to ensure system functionality and enhance overall experience. By leading users to make valid selections and imposing constraints, the system ensures accurate data input, minimizes errors, and ultimately facilitates smoother operation.

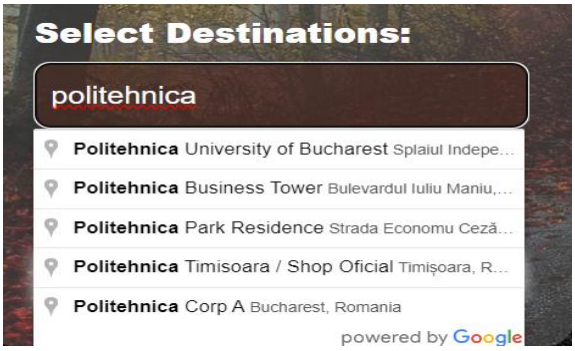


Figure 10. Autocomplete API

Figure 11. Browser Console After Selection

5.5. User Guide

This web application was designed to be accessible for any user, even by those with no technical background. The features and the steps of using the application are described at the top of the *Home* page, so that the user can see them when he first enters the website. Over more, the *Home* page offers general information on Low Emission Zones.

Figure 12. Home Page

As described in this document, and also on the *Home* page, the user would first need to create an account. This only requires name, email and password, as other information is not relevant and would only complicate the process. Right after the account creation, the user is logged in and directed to *My Profile* page. At that point all features are unlocked and users can view and manage the information associated to their account. The web navigation bar, situated at the top of every web page, has been updated as well and allows users to easily navigate across the application. *Figure 13* displays the *My Profile* page of a user that had already registered some vehicles and saved a number of routes.

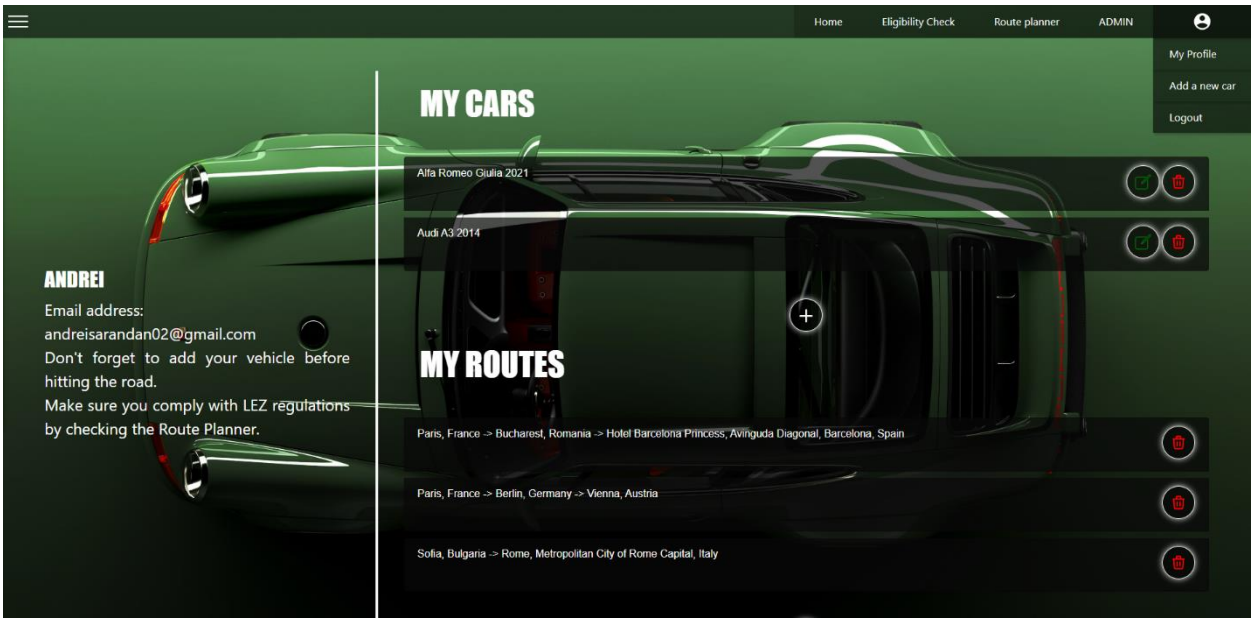


Figure 13. My Profile page

The next step would be registering a vehicle. This can be done from the *Add a New Car* page, that can be accessed either from the *Home* page, by clicking the suggestive image, or from the *My Profile* page. Brand and model are necessary for identification purposes, while the rest of the details are relevant in the Low Emission Zone access validation process. On this page, the user should add any registrations that are already assigned to his car. This can be done through the dynamic form element, which guides the user in the selection process. Any future modifications can easily be made after the car was saved. In the images displayed below, the user is in the process of registering a diesel Euro 4 vehicle, which already has a Germany registration sticker assigned.

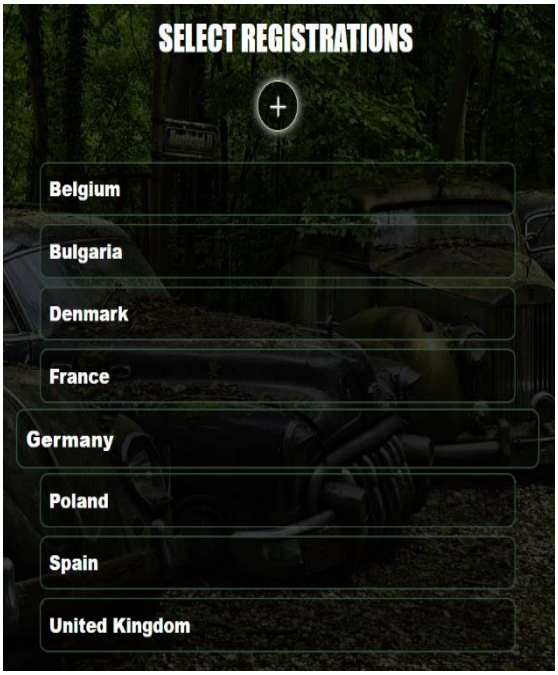


Figure 14. Selecting registration country

Figure 15. Selecting registration type

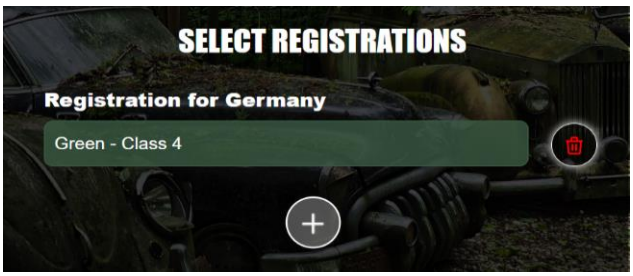


Figure 16. Registration assigned to the vehicle

Route Planner page can be accessed from the navigation bar or from the *Home page*. To fill the form, the user should select his vehicle and add the destinations with the help of Google autocomplete. The number of destinations is completely adjustable. By default, the page loads only one field, but the user can add several. These additional fields can also be removed if necessary. All these actions are guided by intuitive icon buttons. Once the user clicks the arrow button, the form is submitted and he instantly receives the information relevant for Low Emission Zone access in the selected destinations. Additionally, the notifications also suggest that he can use the *Eligibility Check page* to verify his vehicle for potential registrations or compliance with National LEZ schemas.

As an example, the user selects Berlin as the start of the route and Bordeaux as the place to visit. The selected vehicle is the one previously saved (diesel Euro 4 with Umweltplakette 4). Afterwards, he receives two notifications informing him if any Low Emission Zones exist, if he is allowed to access it and which registrations are required. Access in Berlin LEZ is permitted as he already owns the required sticker, but he is lacking Crit'Air 2 sticker that is necessary in Bordeaux. Since he does not own any Crit'Air registrations, he can check which one his vehicle is eligible for, by accessing the link from the notifications.

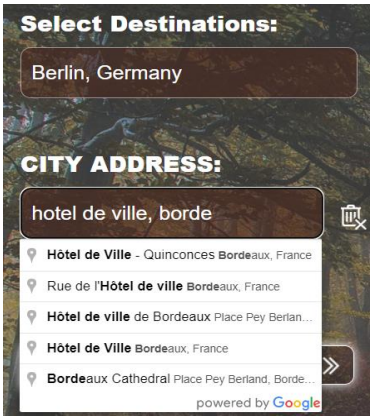


Figure 17. Selected destinations

Figure 18. Notification alerts based on selected destinations

Eligibility Check page can be accessed either from the navigation bar, *Home* page or directly from the notifications received from *Route Planner*. On this page, the user has the possibility to select one of the vehicles saved to his account or to fill in the specifications of another vehicle. This is ideal for users who are considering changing their vehicle or buying a new one. Thus, they can compare the situation, get a better understanding of the regulations, resulting in a better-informed acquisition. Lastly, the user should select the country which he wants to verify for. After selecting the details and clicking the lens-icon button, the user gets an instant reply with the registration eligible for his car and an image of it (where it is applicable), general information of the National LEZ schema and the link of the official regulatory website.

Continuing the example described previously, the user is now testing his vehicle for France registrations and finds out that his vehicle is eligible for Crit'Air 3. Unfortunately, this registration is inferior to Crit'Air 2. Since the user is really keen on visiting France, he tests one of his friend's vehicle as well. Since this vehicle is newer and classified as Euro 5 pollution standard, it is eligible for Crit'Air 1, superior to the registration required, which allows him to enter Bordeaux.

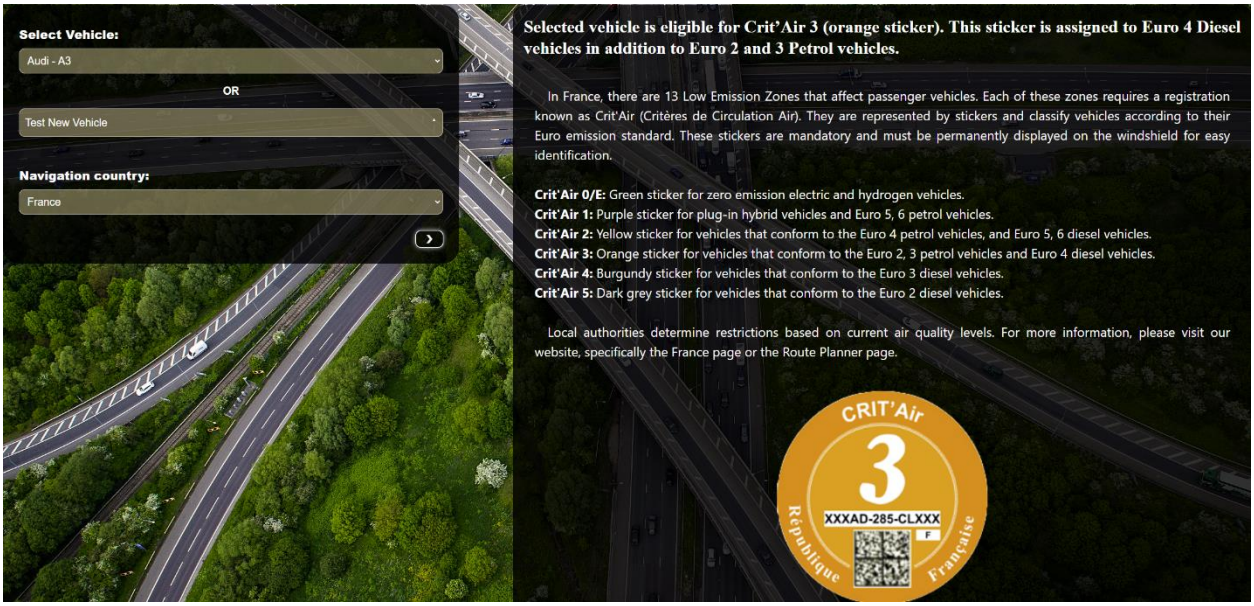


Figure 19. Test diesel Euro 4 saved vehicle

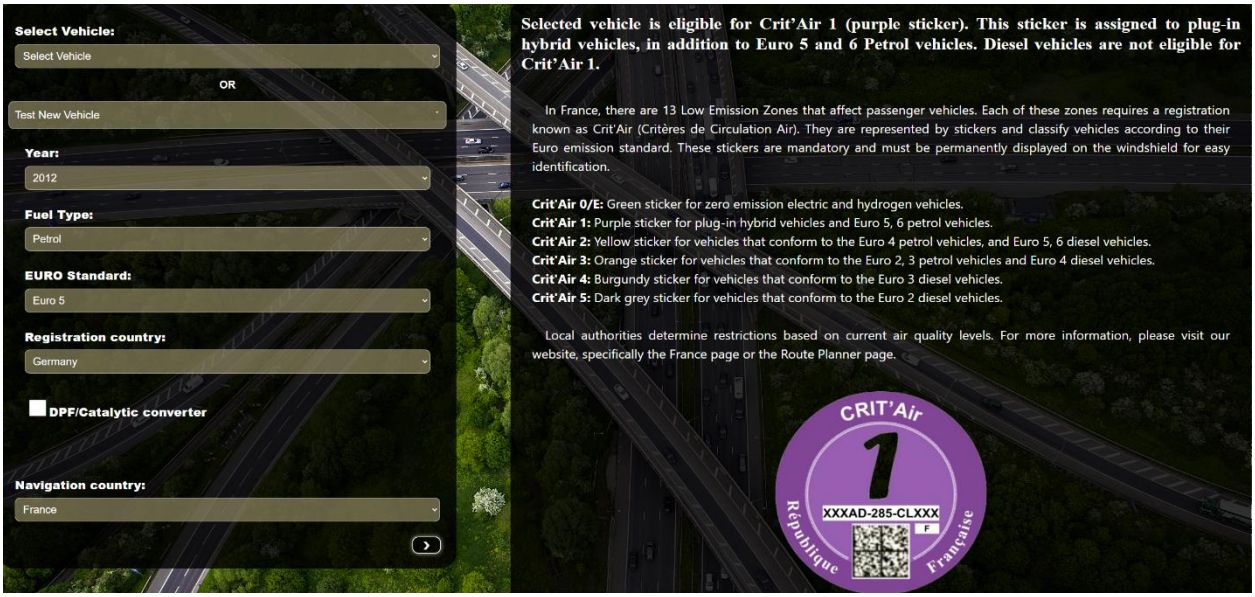


Figure 20. Test new petrol Euro 5 vehicle

Another notable scenario would be if the user wants to visit an Italian city, for example Rome. This prompts the user to enter the period in which he plans to visit, considering that the traffic regulations vary based on time period. In this case, he would be allowed to drive during most part of the year but would be forbidden during the winter when restrictions are raised from diesel Euro 4 to Euro 6. The text of the notification differs a bit from the previous ones, as Rome does not require any registration.

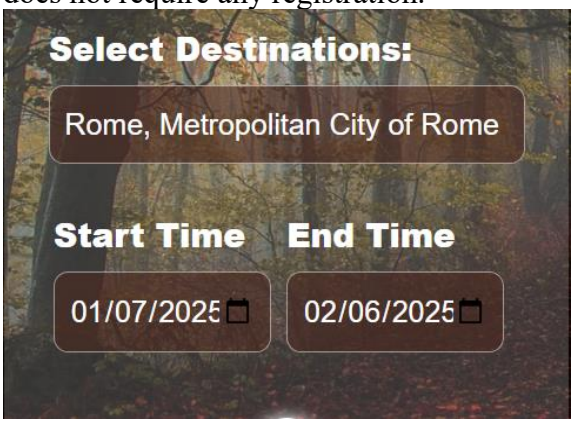


Figure 21. Alert for Rome LEZ

Figure 22. Selection of Italian cities

5.6. Application Testing

The testing phase is a vital part of the software development lifecycle, as it ensures that the system works as designed and meets the requirements set during the analysis phase. This can be done through different techniques and methodologies, depending on the nature of the system. The tests performed to validate this application were both qualitive and quantitative, in order to cover multiple scenarios.

Unit testing is a viable solution for testing the API endpoints and basic functionalities, such as access and authorization for users and data transfer. Python offers the Pytest library, which ensures the possibility to create unit tests for the Flask request handlers [44]. This is done through scripts with predefined input and expected output. The scripts run the different endpoints and verify if the actual output matches the expected one. They operate on the testing environment, which replicates the one used for production (actual usage of the application). This ensures that the original database is not faulted during testing.

Considering the complex nature of this application, the unit tests follow the chronological steps any user would make: create an account and login, save a vehicle, plan a route, test registration eligibility. Without the first steps, the tests cannot target the *Route Planner* and *Eligibility Check* handler. These tests are triggered from the command line and by the end of each run, the script clears the test database.

Integration testing targets the interaction between the components of the application, more specifically, between frontend and backend. It also targets the system's communication with the database and external APIs. This was achieved through the verification of user-input data and its effect in the backend. The integration testing involves two approaches to track the data flow from one module to another. The top-down approach starts the testing from the highest-level module, which as detailed in the *System Architecture* chapter, represents the user interface. The data is submitted through the POST requests and its flow is tracked in the endpoint handlers, the validation algorithms and finally, in the database. The changes in the content or structure of the data must correspond exactly to the changes intended from the start by the user. In the bottom-up approach, the flow of the data is checked starting from the lowest level module, the database. Once again, data integrity is checked on each level, to prevent faulty responses. The bottom-up approach was also used in the application development process.

6. Conclusions

6.1. Discussion and Results

This web application represents a platform that facilitates access to information on environmental zones, where users can benefit from modern technologies to ease the navigation process. Structuring the application in two main web pages, *Route Planner* and *Eligibility Check*, combined with the text-based resources, offers all the information necessary to comply with the regulations. By doing so, the primary scope of mitigating pollution in central areas of urban settlements is achieved.

Route planning becomes more and more difficult because of the current traffic regulations imposed in different environmental zones throughout Europe. These regulations will continue to exist, even though some people may consider them unnecessary. The truth is, as the studies indicate, these restrictions are very effective in mitigating vehicle pollution in urban settlements. The issue lies in the differences between the legal frameworks imposed by European countries. Ideally, online resources help the population learn more about this topic and facilitate their access. But, considering the large amount of information, classic text-based resources are not the best option.

The project presented in this document offers a viable solution, that facilitates access to information about environmental zones, particularly Low Emission Zones. On top of that, the application represents a complex system used for validating vehicle access. It achieves this by

hosting a centralised database model that structures the LEZ restrictions from different countries. The user can register his vehicles in the database as well, which allows the dedicated algorithms to validate its access. Additionally, this application includes methods that automatically decide which registration can be obtained by the vehicle. To improve the efficiency and the functionalities, the application integrates external APIs for air quality data, location autocomplete, geocoding and navigation.

From a functional point of view, the application fulfils all the requirements set in the analysis part. Users can save their vehicles and favourite routes. The endpoint functions of the web system are well configured, ensuring smooth navigation across the different pages. HTTP requests are properly handled by specific functions, which guarantee the user's interaction with the resource. The validation algorithms have been tested and the results confirm they process the data correctly. At the core of the application, the relational database model allows managing and storing all the data. This information is used by the dedicated algorithms and allows dynamic web page rendering. The external APIs used in the development facilitate planning the route when navigating through Europe. Data security is ensured by the password hashing algorithms and authorization methods. The testing process described in the previous chapter ensured the system is functional and works as designed.

Nonfunctional requirements have also been fulfilled by the application. The graphic interface is engaging, as it uses high-quality images to capture the attention of the user, but also opts for dynamic forms and type suggestions to ensure the data accuracy and reduce the use-time. The application facilitates compliance to hundreds of European Low Emission Zones, with the possibility to check multiple cities at the same time and still obtain a fast response. The main advantage of this web application is that users find the information specific for their vehicle, in a fast and concise manner. The algorithms help the user to quickly find the data he requires, without the need to navigate broad text sections. Even so, the information offered is precise and up-to-date, gathered through extensive research, which included official and trustworthy sources.

In conclusion, this application represents a first step towards creating a central hub for information regarding environmental zones and urban traffic regulations. Its architecture allows implementation of future updates, such as adding new schemas or editing already existing ones. Additionally, the administrator dashboard allows technical users to make slight adjustments to the database, in order to keep the information up to date. This application represents a fully functional access validation system, that brings real value and service to the European community.

6.2. Future Plans

In the future, this software application could be updated to gather data and create statistics on the efficiency of environmental zones. In order to do this, the system needs to consider the average vehicle pollutant levels, such as CO_2 (g/km) and particulate matter (g/km). This information is usually registered by the automotive manufacturer in the Certificate of Conformity (CoC) and monitored in yearly car inspections. The necessary data can either be gathered from available emission reports or directly from users. Fortunately, the Python backend would support special algorithms to process this data and create statistics. These statistics would highlight the benefits of environmental zones and potentially draw a conclusion on the more effective frameworks. In order to obtain correct values, the system would require a large number of users.

Scaling the application includes different types of vehicle restrictions and environmental zones, such as Zero-Emission Zones, road tolls or Emergency Air Pollution. These updates further enhance the application's utility and relevance in addressing contemporary transportation

challenges. From a technical perspective, the update would include the addition of new tables for each environmental zone type. The access validation algorithms also need to be extended to cover these regulations.

At the moment, the application is exclusively designed for passenger car drivers. A future expansion could cover all vehicle types, from motorcycles to heavy vehicles. From a database model perspective, such expansion involves the addition of new tables for each vehicle type. Further, it would be required to add new fields in the *Zone* table, to separate the regulations. Also, small adjustments in the validation algorithms, along with the addition of a field where the user selects the automobile class are mandatory. This update not only broadens the application's target audience, but also creates the possibility of collaborating with enterprises for managing vehicle fleets. Also, constant updates would be imperative to ensure that the application remains relevant in time as the LEZ landscape evolves.

Bibliography

[1] European Parliament.(2019, Mar. 22). *CO2 emissions from cars: facts and figures* [Online]. Available: https://www.europarl.europa.eu/topics/en/article/20190313STO31218/co2-emissions-from-cars-facts-and-figures-infographics (retrieved on March 18th, 2024)

[2] European Commission. *Urban Vehicle Access Regulations* [Online]. Available: https://transport.ec.europa.eu/transport-themes/urban-transport/urban-vehicle-access-regulations_en (retrieved on March 20th, 2024)

[3] DieselNet. *Sweden: Environmental Zones Program* [Online]. Available: https://dieselnet.com/standards/se/zones.php (retrieved on May 6th, 2024)

[4] NILU. (2022, Nov. 24). *Premature Deaths Due to Air Pollution Continue to Fall in the EU* [Online]. Available: https://www.nilu.com/2022/11/premature-deaths-due-to-air-pollution-continue-to-fall-in-the-eu/ (retrieved on May 15th, 2024).

[5] P. Holnicki, A. Kałuszko, and Z. Nahorski, "A Projection of Environmental Impact of a Low Emission Zone Planned in Warsaw, Poland", *Sustainability*, vol. 15, no. 23, p.16260, 2023. [Online]. Available: https://www.mdpi.com/2071-1050/15/23/16260 (retrieved on April 2nd, 2024)

[6] DieselNet. *EU: Low Emission Zones (LEZ)* [Online]. Available: https://dieselnet.com/standards/eu/lez.php (retrieved on May 6th, 2024)

[7] The German Emission Sticker. (2023, Aug. 03). *The German diesel ban* [Online]. Available: https://www.germanemissionssticker.com/the-german-diesel-ban/ (retrieved on March 20th, 2024)

[8] Explore France. (2024, Mar. 15). *The Crit'Air anti-pollution vehicle sticker* [Online]. Available: https://www.france.fr/en/holiday-prep/crit-air-anti-pollution-vehicle-sticker (retrieved on March 20th, 2024)

[9] Urban Access Regulations. *Paris Low Emission Zone* [Online]. Available: https://urbanaccessregulations.eu/countries-mainmenu-147/france/paris (retrieved on March 20th, 2024)

[10] C. Holman, R. Harrison, X. Querol, Review of the efficacy of low emission zones to improve urban air quality in European cities, *Atmospheric Environment*, Volume 111, 2015, pp. 161-169

[11] I. Tiseo, " Number of low-emissions zones in Europe 2022-2025, by country", *Statista,* 2023 [Online]. Available: https://www.statista.com/statistics/1321264/low-emission-zones-europe-by-country/ (retrieved on March 25th, 2024)

[12] London Government. (2022, Dec. 14). Usage of Ultra Low Emission Zone (ULEZ) Automatic Plate Recognition Cameras for Crime Prevention and Detection Work by the Metropolitan Police Service [Online]. Available: https://www.london.gov.uk/who-we-are/what-london-assembly-does/questions-mayor/find-an-answer/usage-ultra-low-emission-zone-ulez-automatic-plate-recognition-cameras-crime-prevention-and (retrieved on March 25th, 2024)

[13] B. Lawson and R. Sharp, *Introducing HTML5,* 2nd edition. Berkeley, CA: Peachpit Press, 2012.

[14] N. Ceder, *The Quick Python Book*, 3rd edition. New York, NY: Simon and Schuster, 2018.

[15] Stack Overflow. *2023 Developer Survey* [Online]. Available: https://survey.stackoverflow.co/2023/#overview (retrieved on April 15th, 2024)
[16] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*, 2nd edition. Sebastopol, CA: O'Reilly Media, 2018
[17] Sunbul, "Flask vs. Django: Which Framework Should You Choose?", Red Switches, "Companies Using Flask", 2023, Nov.18 [Online]. Available: https://www.redswitches.com/blog/flask-vs-django/#:~:text=Netflix%3A%20Netflix%20is%20among%20the,Flask%20to%20build%20internal%20tools (retrieved on February 23rd, 2024)
[18] J. Kreibich, *Using SQLite*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2010.
[19] D. Farrell, *Data Management With Python, SQLite, and SQLAlchemy* [Online]. Available: https://realpython.com/python-sqlite-sqlalchemy/ (retrieved on February 19th, 2024)
[20] HG Data. *Companies Using SQLite.* [Online]. Available: https://discovery.hgdata.com/product/sqlite (retrieved on February 23rd, 2024)
[21] D. Crockford, *JavaScript: The Good Parts,* 1st edition. Sebastopol, CA: O'Reilly Media, Inc., 2008.
[22] D. Flanagan, *JavaScript: The Definitive Guide,* 6th edition. Sebastopol, CA: O'Reilly Media, 2011
[23] Amazon Web Services. *What is an API (Application Programming Interface)?* [Online]. Available: https://aws.amazon.com/what-is/api/#:~:text=APIs%20are%20mechanisms%20that%20enable,weather%20updates%20on%20your%20phone (retrieved on May 3rd, 2024)
[24] OMG, *OMG Unified Modeling Language TM (OMG UML)*, Version 2.5.1 , December 2017. [Online]. Available: https://www.omg.org/spec/UML/AboutUML/ (retrieved on May 26th, 2024)
[25] J. Shetty et al., "Review Paper on Web Frameworks, Databases and Web Stack," *International Research Journal of Engineering and Technology (IRJET)*, pp. 5734–5738, 2020. [Online]. Available:https://www.irjet.net/archives/V7/i4/IRJET-V7I41078.pdf (retrieved on June 2nd, 2024)
[26] Amazon Web Services. *What's the Difference Between Frontend and Backend in Application Development?* [Online]. Available: https://aws.amazon.com/compare/the-difference-between-frontend-and-backend/ (retrieved on May 3rd, 2024)
[27] Flask. *Blueprints and Views* [Online]. Available: https://flask.palletsprojects.com/en/2.3.x/tutorial/views/ (retrieved on May 6th, 2024)
[28] G. Svennerberg, *Beginning Google Maps API 3*. New York, NY, USA: Apress, 2010.
[29] Open Cage. *Comparing and testing geocoding services* [Online]. Available: https://opencagedata.com/guides/how-to-compare-and-test-geocoding-services (retrieved on June 2nd, 2024)
[30] WAQI. *World Air Quality Index* [Online]. Available: https://waqi.info/ (retrieved on May 10th, 2024)
[31] Amazon Web Services. *What is an IDE (Integrated Development Environment)?* [Online]. Available: https://aws.amazon.com/what-is/ide/#:~:text=An%20integrated%20development%20environment%20(IDE,easy%2Dto%2Duse%20application (retrieved on May 3rd, 2024)

[32] Conda. *Conda Documentation* [Online]. Available: https://conda.io/projects/conda/en/latest/index.html (retrieved on April 25th, 2024)
[33] IBM. *What is a REST API?* [Online]. Available: https://www.ibm.com/topics/rest-apis#:~:text=A%20REST%20API%20(also%20called,transfer%20(REST)%20architectural%20style (retrieved on May 3rd, 2024)
[34] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces,* 2nd edition. Sebastopol, CA: O'Reilly Media, Inc., 2012
[35] D. Peng, L. Cao, and W. Xu, "Using JSON for data exchanging in web service applications," *Journal of Computational Information Systems*, vol. 7, no. 16, pp. 5883-5890, 2011.
[36] A. T. Holdener III, *Ajax The Definitive Guide*. 1st edition, O'Reilly Media, 2008
[37] C. Xia, G. Yu, and M. Tang, "Efficient implement of ORM (object/relational mapping) use in J2EE framework: Hibernate". *International Conference on Computational Intelligence and Software Engineering*, 2009, pp. 1-3 [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5365905 (retrieved on May 26th, 2024)
[38] T. Chen, et al., "An empirical study on the practice of maintaining object-relational mapping code in java systems." *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2901739.2901758 (retrieved on May 26th, 2024)
[39] D. Gaspar, and J. Stouffer, *Mastering Flask Web Development: Build Enterprise-grade, Scalable Python Web Applications, 2nd edition*. Birmingham, United Kingdom, Packt Publishing, 2018.
[40] Ioana Fagarasan, *Analiza Si Proiectarea Sistemelor Informatice*, Universitatea Politehnica din Bucuresti, Facultatea de Automatica si Calcuatoare, Ingineria Sistemelor, Curs Anul 4.
[41] H. S. Oluwatosin, "Client-server model", *IOSR Journal of Computer Engineering*, vol. 16, no. 1, pp. 67-71, 2014.
[42] I. Permana, M. Hardjianto, and K. Ahmad Baihaqi, "Securing the Website Login System with the SHA256 Generating Method and Time-based One-time Password (TOTP)", *Systematics Journal*, vol. 2, no. 2, pp. 65–71, Aug. 2020. [Online]. Available: https://journal.unsika.ac.id/index.php/systematics/article/view/3756 (retrieved on April 3rd, 2024)
[43] J. Kolce, M. Kroger, I. Curic, S. Saeed, J. Mott, M. D. Green, and C. Buckler, *JavaScript: Best Practice*, SitePoint Pty Ltd, 2018.
[44] B. Okken, *Python Testing with pytest*, 2nd edition. Pragmatic Bookshelf, 2022.