|  |  |  |
| --- | --- | --- |
| A blue and black logo  Description automatically generated | **Departamentul Automatică şi Informatică Industrială**  **Facultatea Automatică şi Calculatoare**  **Universitatea POLITEHNICA din Bucureşti** | A blue and black logo  Description automatically generated |

LUCRARE DE LICENŢĂ

**Web Application for Environmental Zones Awareness and Route Planning**

Coordonator Absolvent

Prof. Anca Daniela Ionita Andrei SARANDAN

2024

Contents

[Contents 2](#_Toc167629920)

[Part I - General Aspects 5](#_Toc167629921)

[1 Introduction 6](#_Toc167629922)

[2. Inclusion of the study's domain of interest 7](#_Toc167629923)

[2.1 Urban Vehicle Access Regulations 7](#_Toc167629924)

[2.2 Low Emission Zones 7](#_Toc167629925)

[2.3 LEZs throughout Europe 8](#_Toc167629926)

[2.3.1 Germany and France 8](#_Toc167629927)

[2.3.2 Italy and Spain 9](#_Toc167629928)

[2.3.3 Northern Europe 9](#_Toc167629929)

[2.3.4 Eastern Europe 9](#_Toc167629930)

[2.4 Current situation in Europe 9](#_Toc167629931)

[2.6 Forecasts 10](#_Toc167629932)

[2.8 Similar applications 11](#_Toc167629933)

[2.8.1 Green-Zones.eu 11](#_Toc167629934)

[2.8.2 Urban Regulations 12](#_Toc167629935)

[3 Technical Documentation 13](#_Toc167629936)

[3.1 Software Technologies 13](#_Toc167629937)

[3.1.1 Hypertext Markup Language 13](#_Toc167629938)

[3.1.2 Python 14](#_Toc167629939)

[3.1.3 Flask 14](#_Toc167629940)

[3.1.4 SQLite 15](#_Toc167629941)

[3.1.5 JavaScript 15](#_Toc167629942)

[3.1.6 APIs 16](#_Toc167629943)

[3.1.7 UML 17](#_Toc167629944)

[3.2 Backend Development 18](#_Toc167629945)

[3.2.1 Foundations 18](#_Toc167629946)

[3.2.2 Route handling in Flask 18](#_Toc167629947)

[3.2.3 Database 19](#_Toc167629948)

[3.2.4 Access validation algorithms 21](#_Toc167629949)

[3.2.5 Dynamically rendered web pages 23](#_Toc167629950)

[3.3 Frontend Development 23](#_Toc167629951)

[3.3.1 General aspects 23](#_Toc167629952)

[3.3.2 Maps API 24](#_Toc167629953)

[3.3.3 OpenCage Geocoding API 24](#_Toc167629954)

[3.3.4 Air Quality API 24](#_Toc167629955)

[3.3.3 Asynchronous function for handling form submission 25](#_Toc167629956)

[3.3.4 Notification display 25](#_Toc167629957)

[3.3.5 Dynamic elements 26](#_Toc167629958)

[3.3.6 Conditional selection 27](#_Toc167629959)

[3.4 Software Development Moldes and Tools 28](#_Toc167629960)

[3.4.1 Integrated Development Environment (IDE) 28](#_Toc167629961)

[3.4.2 Virtual Environment 28](#_Toc167629962)

[3.4.3 Database Management Tools 29](#_Toc167629963)

[3.5 Development Techniques 29](#_Toc167629964)

[3.5.1 REST APIs 29](#_Toc167629965)

[3.5.2 JSON 30](#_Toc167629966)

[3.5.3 CRUD in Web Applications 30](#_Toc167629967)

[3.5.4 AJAX 31](#_Toc167629968)

[3.5.5 Document Object Model 31](#_Toc167629969)

[3.5.6 ORM 32](#_Toc167629970)

[3.5.7 Authentication and Authorization 32](#_Toc167629971)

[3.5.8 Templating Engine 33](#_Toc167629972)

[PART II – PROJECT CONTRIBUTIONS 35](#_Toc167629973)

[4 Project Motivation 35](#_Toc167629974)

[4.1 Addressing the Complexity of Navigation throughout LEZs 35](#_Toc167629975)

[4.2 The Recommended Solution 36](#_Toc167629976)

[5 Development Methodology 36](#_Toc167629977)

[5.1 Analysing Requirements and Specification 36](#_Toc167629978)

[5.1.1 Information Sources 38](#_Toc167629979)

[5.1.2 Application Use Cases 39](#_Toc167629980)

[5.1.3 Access validation process 40](#_Toc167629981)

[5.1.4 Non-functional Requirements 41](#_Toc167629982)

[5.1.5 Application Constraints 42](#_Toc167629983)

[5.2 Design elements 42](#_Toc167629984)

[5.2.1 System architecture 42](#_Toc167629985)

[6 Description of the Practical Application 44](#_Toc167629986)

[6.1 User Guide 44](#_Toc167629987)

[6.2 Application Testing and Validation 45](#_Toc167629988)

[6.3 Results Obtained 46](#_Toc167629989)

[6.4 Future Plans 46](#_Toc167629990)

[7 Conclusions 46](#_Toc167629991)

[Bibliography 48](#_Toc167629992)

[Appendix 1. Code Snippets 50](#_Toc167629993)

# Part I - General Aspects

# 1 Introduction

The World Wide Web represents without a question an indispensable tool in the daily life during present times. On the other hand, the escalating threat of global warming is more prominent than ever before, with recent events occurring across the globe. Ideally, the internet’s capabilities must 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 Low Emission Zones (LEZs). This web application seeks to raise awareness and trust in this scope, enabling users to make informed planning and consider the environmental impact of their travel and vehicle choice. It represents more than just a text repository or a presentation website as it is a comprehensive platform, which includes modern technologies like Application Programming Interfaces (APIs), extensive database model and complex access validation algorithms. All of these aim to deliver a great user experience, from account creation processes, thus raising the possibility of personal vehicle registration, to trip planning features. To improve the user experience, all the complex logic methods, data sets and algorithms are hidden behind the friendly and interactive interface.

The creation of this web application was done in three main steps. Firstly, the application development was based on a tight connection between backend and frontend parts, involving different programming languages and frameworks. Secondly, the extensive relational database model was designed to store relevant information about both the Low Emission Zones and application users. Lastly, in creating the database, a prolongated research was needed to cover the different complexities of the hundreds of LEZs throughout Europe. A key factor was to also fit the everchanging character of the LEZs. The information presented on the web application must be correct, up-to-date, and presented in an understandable manner. By doing so, the web application aims to contribute to broader efforts to create a more sustainable and resilient future for all.

The scope of this thesis revolves around developing an application geared towards improving user experience in navigating Low Emission Zones (LEZs), with a focus on providing information and validating access within LEZ cities. Since this concept is very new and still in development, the main objective is to raise awareness and promote their acceptance. LEZs may pose complexities for many individuals, hence efforts are needed to simplify comprehension and facilitate access.

The aim is to create an intuitive web application that provides information and help to access these Low Emission Zones throughout Europe. Since this is an ever-changing topic, the database system is incorporated to cover the regulations that are effective in July 2024. It is designed so that it can be adapted to ongoing updates, in order to provide valid information to the end-user.

# 2. Inclusion of the study's domain of interest

## 2.1 Urban Vehicle Access Regulations

The increasing concerns regarding pollution necessitate the implementation of various measures aimed at enhancing air quality. Pollution, resulting from a multitude of sources and presenting in various forms, possesses 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 concerning 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]. From all means of transports cars represent the most used vehicle, accounting for 60.6% of the total emissions disposed by transport, with an average of 1.6 cars per European habitant. Passenger cars stand out as a major polluter, accounting for 61% of total CO2 emissions from EU civil transport [1]. The European Union is striving to drastically reduce air pollution by implementing new legislation aimed at pushing new vehicles towards achieving zero CO2 emissions. However, the actual progress has proven to be much slower than it was hoped.

The primary methods of reducing CO2 emissions from vehicles are manufacturing more efficient cars and changing the fuel type. This means transitioning to more eco-friendly ones, ideally derived 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], 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, changing parking rules and limiting traffic in certain areas. The goal is to reach better air quality standards, while also improving traffic [3].

Generally, vehicle categorization and access are based on the vehicle type, fuel type 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

Low Emission Zones (LEZs) are geographical areas, usually located in large urban settlements, where specific restrictions are 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 such as NO2, which are highly correlated to several respiratory diseases.

The concept of environmental zones originated in Sweden in 1996 and were 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.

In Europe, the three main air pollutants of concern are represented by nitrogen dioxide (NO2), particulate matter (PM, also referred to as PM10 or PM2.5 depending on the particle diameter) and ozone. The rationale behind the functionality of Low Emission Zones is to restrict as much as possible the access of highly polluting vehicles. This can be achieved either by raising the costs of access through fees or tolls or by completely forbidding the access. Another approach would be retrofitting older vehicles to meet modern regulations, by adding diesel particle filters (DPF). These filters could elevate the vehicle to a slightly higher emission standard [5].

## 2.3 LEZs throughout Europe

### 2.3.1 Germany and France

Considering the transformative nature of a Low Emission Zone, to impose restrictions on vehicle access in central areas, such a solution cannot be swiftly implemented. However, these measures are considered more and more important because of their effectiveness in reducing pollution and CO2 emissions in urban settlements.

The European Union proposes a strategy to gradually implement such areas in the following years [4]. This approach facilitates the acceptance and adoption of a Low Emission Zone and allows each nation to fine-tune the schemas to better fit the needs and possibilities of the country and its citizens. Local authorities can effectively manage the requirements and priorities, enhancing the efforts to minimize pollution as much as possible. Because of this, the Low Emission Zone term can be found under a different name, depending on the national law it has been regulated by. 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 and criteria that separate vehicles based on Euro standard. While there is no clear recipe for implementing a Low Emission Zone in a big urban settlement, giving free-will for the nations to create and personalize theirs, has led to the existence of many such forms. Having more variations of the same regulation results in more case-studies and faster evolution in this field. In a relatively short span, regulations have taken on many different forms, some being more effective than other and some being easier to implement than other. Even though each approach presents its own set of merits and drawbacks, the end objective remains the same.

For example, one of the main differences between the Low Emission Zone regulations in Germany and France lies in the criteria selected for classification. To avoid confusion, each vehicle is assigned a sticker, which must be displayed on the windshield for enforcement by authorities. In Germany, vehicles can be assigned one of the three available emission stickers based on their Euro emission standards. Green represents the cleanest and most environmentally friendly out of them all, followed by yellow and, lastly, red for the highest emissions. 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 be eligible for the green sticker [6]. 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 and hydrogen vehicles, to dark gray for the most polluting vehicles.

While both systems serve a similar purpose of regulating vehicle emissions within urban areas, the specific criteria for classification and the appearance of the stickers are quite different. The Crit'Air vignette also includes a long-term strategy to introduce this schema throughout the cities with more than 150.000 inhabitants [7]. Additionally, the plan is to gradually tighten the standards as the years advance, by switching the minimum required sticker for access to the next one. As an example, in Paris, regulations in place since 1 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 on the 1st of July 2023, but it had to be postponed [8].

### 2.3.2 Italy and Spain

### 2.3.3 Northern Europe

The standards and regulations from the Northern European countries vary a lot from the scenarios described above. This situation adds another layer of complexity for individuals seeking to access Low Emission Zones.

Compared to the stickers present in countries like France and Germany, the European Low Emission Zone schema is further compounded by a different strategy in Northern Europe. Nations in this region, such as Denmark (Miljøzone), Sweden(Miljözon) and the Netherlands(Milieuzone) implement their own systems for emission regulations. Even though rationale remains the same, to reduce pollution by restricting the highly polluting vehicles access in city centres, the minimum requirements for entering Low Emission Zones in these regions are different.

First of all, sticker-based registrations do not exist and foreign ones do not apply. Each nation sets the minimum requirement for combustion-based vehicles and all 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 process of complying for the citizens, therefore possibly increasing the efficiency of the LEZ. However, foreign vehicles need to register online, typically prior to entering the LEZ to avoid penalties. This registration only allows access int the LEZs for vehicles that actually comply with the minimum emission class requirements set by local authorities.

Northern Europe Low Emission Zones are primarily enforced through smart-camera systems, designed to capture the vehicle’s registration number before entering the designated areas. This system captures the registration plate number and accesses the central database to verify the vehicle’s eligibility for entry. This automated approach enhances efficiency and accuracy while reducing reliance on and costs for manual inspections.

### 2.3.4 Eastern Europe

## 2.4 Current situation in Europe

The different scenarios derived from the LEZ schemas described in the previous chapter underline the difficulty of understanding and respecting the specific requirements and regulations in place foreach country. This further complicates the experience for both residents but especially for tourists which necessitate careful planning for compliance measures when traveling across borders.

Considering the aforementioned context, travelling through Low Emission Zone becomes a significant challenge for the average individual. The multitude of European countries that have adopted a form of Low Emission Zone and the diverse range of regulations taken into consideration create a complex navigation landscape.

The European Union legislation encourages member states to establish their own regulations for Urban Vehicle Access Regulations, such as Low Emission Zones (LEZs). National authorities must take into consideration different factors such as population density, existing infrastructure, public transport etc. This decentralized structure offers several advantages. One of them is the fact that it facilitates the implementation of diverse strategies, providing a range of test cases to evaluate the efficacy of different models in addressing air pollution and promoting sustainable urban mobility. It is important to acknowledge that national governments are best positioned to understand their own environmental challenges and propose efficient solutions accordingly.

However, this system can also lead to disorder in the mind of the population because of the abundance of legislation. With regulations varying, not only between countries, but also among different municipalities within the same country, drivers may encounter various requirements as they navigate through different regions which can lead to confusion and logistical challenges, particularly for individuals who frequently travel between areas with differing regulations.

The Crit'Air sticker system in France and the Umweltplakette system in Germany are a good example for this diversity. This case involves nine stickers utilized across two countries, each of them available exclusively within its respective jurisdiction. The only similarity between the two schema is the fact that they are color-coded stickers. This variety can pose challenges for most people.

## 2.6 Forecasts

According to the report made by statista.com [9], forecasts expect a significant expansion of Low Emission Zones (LEZs) throughout Europe. Projections indicate a substantial increase in the number of LEZs, with an estimated total of 510 zones across 17 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]

The multitude of terms used to designate Low Emission Zones (LEZs) across Europe drives up the complexity for regular individuals when navigating these environmental zones. While the purpose for each designation remains to reduce emissions and improve air quality, the linguistic and cultural diversity adds layers of intricacy.

From "Umweltzonen" in Germany to " Miljøzone " in Denmark, and " Distintivo Ambienta" in Spain, the wide terminology presents a challenge for users seeking clarity and understanding. This complexity extends with regulations implemented in France, Belgium, England, Denmark, Sweden, Norway, Hungary, and Italy, each with its unique terminology and approach.

The effectiveness of Low Emission Zones is directly linked to both the rigor of the implementation but also the level of compliance. While the establishment of a LEZ is a great proactive first step towards improving air quality in urban settlements, in reality, the impact relies on the degree individuals approve and respect the regulations enforced in such areas.

It is essential to keep in mind that even the best designed LEZ strategy can fail to reach it’s intended goals if control measures are lacking or if vehicle owners do not comply. As The Deputy Mayor for Transport has confirmed in [10], automated monitoring systems, such as the use of plate recognition cameras, prove to be an effective enforcement strategy.

Over more, in order to build popularity and trust in any form of Urban Vehicle Access Regulations, public awareness campaigns serve a crucial role in ensuring that LEZ regulations are upheld. All Nations need to pay effort into convincing the public that the only objective is not to create inconveniences or to impose financial burden, 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.

For individuals to fully comply and embrace the notion of Urban Vehicle Access Regulations imposed through Low Emission Zones, it is imperative to be provided a clear overview of the existing LEZ structures and their benefits. The large number of LEZs across Europe, navigating across the continent, throughout major cities, can become challenging and may create confusion and inconvenience for drivers.

## 2.8 Available Applications

### 2.8.1 Green-Zones.eu

The Green Zones application provides comprehensive information regarding the Low Emission Zones schemas across Europe. The application is split on both mobile and the web platform, providing valuable resources for individuals seeking to understand and comply to the regulations in LEZs.

The mobile app offers the users the possibility to conveniently add relevant details about their vehicle. Then, in order to check compliance with a LEZ, the user needs to select one of the points highlighted on the map. This feature is offered free of charge and accessible to all users. Additionally, the app provides air quality information on a number of cities, which is an important factor taken into consideration when evaluating the efficiency of a LEZ.

However, to access additional features and services, like saving different vehicles to an account, users are required create a subscription-based account, which require a monthly fee. While the premium services offer added functionality, this is rather useful for enterprise users.

For regular users who only plan a few trips each year, the associated subscription fee could cause significant drawbacks. While the app offers valuable features and services, like the vehicle registration, the app only promises the latest data on for the three available subscription plans. In many cases, the subscription fee may outweigh the benefits for individuals that rarely travel through Low Emission Zones. This subscription-based model may discourage casual users who seek to information regarding LEZ regulations compliance.

The web platform is a great complement to the mobile app and is designed to provide comprehensive information about Urban Vehicle Access Regulations, such as LEZs, actual environmental regulations. The website can also be used to purchase registrations for several countries which have LEZ schemas active.

Overall, the Green Zones app offers a user-friendly interface and valuable features for navigating LEZs, thereby promoting environmental awareness. However, the subscription-based model may present a barrier to entry for some users, particularly those who do not require frequent access to premium features.

### 2.8.2 Urban Regulations

UrbanAccessRegulations.eu is designed as a comprehensive online platform for information on different urban access regulations throughout Europe. A significant advantage is the fact that the website offers a wealth of information, organized in an intuitive manner, accessible to anyone with internet access. The website covers the complete range of regulations, from Low Emission Zones (LEZs) to road tolls and emergency schemas, creating a place to find relevant information for any situation.

One notable feature of UrbanAccessRegulations.eu is the extensive and well-structured database. Users can navigate easily across different categories to find the information relevant to their specific situation or curiosity. Over more, the website offers very detailed information for different categories of vehicles, which assures accessibility for a wide range of users.

However, while the coverage offered is very large, the UrbanAccessRegulations.eu may overwhelm some users with the extensive volume of information at hand. In spite of the well-organized structure, users might need to spend long periods of time diving deep into the website to find the details specific for their needs. Further, the addition of personalization features, such as saving vehicle details and registrations, would expend the possibilities of the website's appeal to users. The large variety of vehicles and Urban Access Regulations could be better structured to be displayed according to individual circumstances.

Another positive note, UrbanAccessRegulations.eu offers the information and services free of charge, available online to all users regardless of their financial resources. This free of cost model creates a tremendous advantage, as it eases access to crucial information about Low Emission Zones. Providing so much information at no cost, the website drops financial burdens and facilitates the learning about LEZs. This serves as a fundamental step towards fostering compliance. As individuals become more informed about LEZs, they are better prepared to adhere to regulations, therefore improving the effectiveness of the already existing LEZs and potentially creating a social context for the establishment of additional zones in the future.

In summary, UrbanAccessRegulations.eu provides a large number of valuable resources for users seeking information on urban access regulations throughout Europe. While the offer compounds comprehensive coverage and is free to use, the website's diverse volume of information and less user-friendly navigation may pose challenges for some unexperienced web users.

# 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 interoperability between different platforms and software systems. Some of the well-known languages are HTML (Hypertext Markup Language) and XML (Extensible Markup Language) which 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. Typical for markup languages, HTML syntax is structured based on elements created by tags and key words inserted inside them to indicate the way of displaying the text in the web browsers. The tags are enclosed within angle brackets (</>) and contain attributes which specify additional characteristics or information about the content.

In web development HTML is used to define the structure and layout of the page. By creating an HTML document, developers compose functional documents, using the specific elements such as divs, paragraphs, spans, headings, lists, forms, images, anchor tags, etc. Having a clear and concise syntax, HTML facilitates the process of creating well-structured web pages that can be interpreted by most web browsers and across different devices and operation systems. From an architectural perspective HTML can be viewed as the foundation, used to create a strong product. For improving the visual appearance and to create interactive web applications, HTML lays the possibility for integrating dynamic features built by other technologies like 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 by HTML5 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, using the tag elements like <video> or <audio>, but also brings advanced controls for forms and APIs for geolocation [13]. This latest iteration of HTML allows developers to leverage their skills in order to create interactive online applications, implementing a seamless user experience across different devices.

### 3.1.2 Python

Python is a very powerful programming language known for versatility and reliability, making it an excellent choice for a wide selection of applications. Its simple syntax is very rather similar to the English language, at least compared to other programming languages. Another advantage of Python it the extensive standard library and wide range of third-party packages. This allows developers to tackle diverse tasks from data analysis to artificial intelligence and automation to web development. The disposal of different tools used for different situations improves their efficiency.

As mentioned, one of Python's key aspects lies in its flexibility and ease of use, allowing developers to quickly test ideas before implementing fully functional solutions. The clean and intuitive syntax, combined with typing extensions generally improve code readability and maintainability, making it accessible to both junior and senior programmers. Over more, Python benefits of automatic memory management which simplifies the development, reducing the time and resources required to build and manage complex software systems.

When it comes to web development, Python offers several strong and scalable backend solutions. The main frameworks for web are Flask and Django, which provide developers with powerful tools for creating secure web applications features. Over more, Python's extended support for database integration and REST APIs makes it suitable for the backend development of complex software applications. These frameworks ensure seamless interaction with the frontend even if it is implemented using external interfaces and services.

Overall, Python's versatility and large ecosystem make it 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 and lightweight Python framework, efficient in web development and allows developers to quickly implement the fundamentals of their applications. It was developed by Armin Ronacher and became popular for its minimalist design philosophy that provides features to build full-scale web applications in a relatively short time, while also reducing the redundancy code fragments. At its core, Flask framework disposes of the essential tools for handling HTTP methods, routing requests, dynamically render templates and also ensuring the scalability of the system.

Integrating the different technologies needed 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. Despite the need to plan meticulously, the minimalist design of Flask applications does not constrain developers with tight coding patterns and allows them to focus their resources on innovation and problem-solving.

Making use of Python's asynchronous programming capabilities, Flask applications efficiently handle concurrent requests, while also assuring optimal performance even under heavy loads [11]. 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 a different industries and business sectors, from small businesses and startups to large enterprises. The short development time needed to construct a demo application and the versatility make it 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 [15] but it’s also widely used by microblogging platforms, content management systems (CMS), REST API development and data analysis platforms. Flask's extensibility and modularity make it suitable for projects of any size and complexity, offering developers the flexibility to gradually scale and evolve their applications when needed. Overall, Flask's flexibility, simplicity, and performance make it a popular choice among software developers and enterprises seeking to build modern and efficient web applications.

### 3.1.4 SQLite

SQLite is a self-contained, lightweight and serverless C library for relational database management that is widely acclaimed for its reliability and versatility. 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 [18], indexes, and triggers. The database can be accessed using a custom variant of the classic SQL query language.

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 SQLite tables in database of the applications, creating a seamless interaction between the application's codebase and the database [20]. Over more, popular third-party libraries such as SQLAlchemy provide advanced ORM features, further enhancing the flexibility and functionality of Python applications using SQLite databases.

Furthermore, SQLite's smooth compatibility with Python makes it an ideal choice for many developers. Python's built-in support for SQLite offers developers a strong foundation for database management. While SQLite integration is provided within Python's standard libraries, leveraging its functionality also requires deep understanding of SQL queries, which are adapted to SQLite syntax. The integration necessitates meticulous attention to detail in the creation of SQL statements or when managing database connections to ensure data security and integrity. Despite its apparent simplicity, utilizing SQLite in the integration of databases in Python based applications is very effective.

Overall, SQLite's combination of simplicity and reliability makes this combination with Python a great choice for developers seeking a powerful and scalable database solution for their applications. The support for class modelling which can seamlessly be integrated with Python’s OOP concepts, further solidify its position as one of the most preferred database technologies. SQLite in the software development landscape, being actively used by many enterprises [20].

### 3.1.5 JavaScript

JavaScript 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 [20.5].

Over the years, a multitude of libraries and frameworks have emerged to further augment JavaScript's capabilities. Among the most popular, React.js [21], maintained by Meta, has gained widespread adoption for its component-based architecture and efficient Document Object Model manipulation, making it a popular choice for front-end developers. On the other hand, Angular, maintained by Google, is 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 [22]. 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), remains a solid option for many applications. Its complex features yet flexible approach allows the creation of customized solutions without the overhead of additional dependencies, promoting efficient development practices.

JavaScript's versatility shines through its seamless integration with various APIs. This way, developers can access 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 APIs to manipulate the Document Object Model, JavaScript serves offers robust solutions for building interactive web applications with modern features.

Asynchronous programming [23] 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, the JavaScript manages effectively 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, therefore it is considered an important tool in the development of modern, interactive web applications.

The architecture of JavaScript is event-driven and creates the possibility to handle multiple events simultaneously, further enhancing the feasibility for intuitive and functional user interfaces. The creation of event listeners attached to different elements of the web page, cover all the possible scenarios of user interactions, so the focus on the developer shifts on modelling system behaviour.

Generally, the syntax is considered lightweight and code structure can be adapted to specific needs. Many IDEs offer good JavaScript integration and code auto complete which contribute to the appeal of this language to software development community. The amount of code necessary for creating basic elements is relatively short, creating the possibility to swiftly develop the prototype of an application. The flexible architecture allows to iterate rapidly when exploring different approaches and refining the code before releasing the product that meets production-ready requirements. JavaScript programming language adheres to modern web standards while also ensuring cross-platform compatibility, which make it an ideal choice for creating web applications.

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 and versatility combined with the 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 APIs

Application Programming Interfaces (APIs) are sets of rules or mechanisms created by programming code which function as intermediaries for different software applications to interact and communicate with each other [25] [26]. These rules define the methods through which different software components can interact with each other. Each side involved in the communication follows the predefined protocols or request and sending data. APIs are fundamental in contemporary software engineering as they are used by developers to integrate different components, to connect a multitude of services, tools, and functionalities. The creation and usage of APIs follows predefined protocols in order to ensure consistency and security of the data in the communication.

One of the key advantages that comes with the usage of APIs represents the abstraction of underlying and complex processes. By doing so, software engineers can put their focus on developing new functionalities without the need to iterate over all the functional processes behind those features. As the name suggests, APIs compose an interface that allows developers to skip the complexities of software programs that are not relevant for their tasks. By this, the integration of third-party services becomes much more effective and simpler.

Additionally, APIs can be used to create modular and scalable software architectures. This is achieved by breaking down large systems into smaller manageable components. These elements are easier to develop and manage. With this modular approach the focus is shifted on building the individual components set to complete different tasks. Afterwards, the system is built by interconnecting these elements through APIs.

A Web API, also known as a Web Service API, serves as the protocol for communication between the web server and the browser. This conducts the data exchange and communication between the two. In general, all web services are considered APIs. However, not all APIs are specifically designed for web systems. Among the different types of API types, the REST APIs are commonly used for web development and adhere to a standard architectural style [25] [27]. In fact, they have become so widespread that in the context of modern web development, the term API alone is used for referring to REST APIs specifically. This monopole over the industry reflects the shift towards web-based architectures. Therefore, the RESTful principles have become predominant in contemporary application development.

In general, REST APIs facilitate online communication via HTTP requests for performing standard database operations such as creating, reading, updating, and deleting records within the server resource. The well-known HTTP methods: GET, POST, PUT, and DELETE [29] are used for these operations. Data is delivered through the requests and structured in various formats including JSON, XML, or plain text. The headers of the requests are very important for the interactions REST API. They contain important identifier information such as metadata, authorizations, URIs, caching, and cookies.

Overall, APIs represent one of the most used tools in modern software development, as they are information in system interoperability scalability. They act as rules or contracts which define how different software applications interact. This way, available software services and resources can also be integrated by emerging ones, as the complexities are abstracting away. By defining standardized communication protocols, APIs ensure that software components transfer data effectively, regardless of the platforms or technologies behind them.

### 3.1.7 UML

UML stands for Unified Model Language and represents a standardized set of diagrams which are meant to help both software developers and stakeholders with the construction and the visualization of the system. This modelling language offers blueprints for a wide range of infrastructures, which reflect different perspectives on the parts or usage scenarios of the application. These diagrams can be views both from a technical or from a business perspective. UML includes structural diagrams to represent the static aspects, behavioural diagrams to represent the modelling of the system elements and diagrams for representing the interactions between its elements.

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, as newcomers can quickly get an understanding of the different aspects of the system.

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 lifecycle. The features of UML are extended by other modelling languages, like SysML (Systems Modelling Language), which was specifically designed to include complex hardware and software elements used in system engineering.

## 3.2 Backend Development

### 3.2.1 Foundations

### Backend development represents the backbone of any web application’s architecture and it can be considered the engine that powers the entire system. It is responsible for a range of critical activities, like logic implementation, data processing, and handling communication with the user interface. A robust backend is essential for applications intended to fulfil a large purpose, empowering them to manage data effectively and to interact with users. Without the complex backend, web applications are essentially limited to static website pages, lacking the dynamic functionalities available in most modern digital experiences.

### In the context of this application, the backend architecture carries a crucial importance, as it is responsible for handling data management, LEZ access validation through algorithms, user account creation, HTML requests and responses. One of the key aspects of the backend architecture in a web application is the establishment of routes which act as pathways for directing incoming web requests to the appropriate resources within the application. Managing a meticulous route configuration, ensures smooth user interaction and navigation within the web application.

### 3.2.2 Route handling in Flask

Route handling represents a fundamental aspect of web backend development, as it is responsible for dictating how URLs are mapped to specific functions within the application. Considering that Flask is the main framework used in the development of this application, this mapping enables the execution of specific code sequences when a user accesses a particular endpoint. In Flask applications routes are generally managed within the views.py file where they are defined by special functions.

These functions are written in standard Python code. They define the behaviour expected for the system and the algorithms triggered when the user accesses a specific URL. Also, they are declares using Python function decorators. Specifically for web frameworks, the decorators take two main parameters. First, they take a string parameter that reflects the URL or endpoint served by the function. The second is the *methods* parameter, which is a list of strings and refers to the type of HTTP requests handled in the route.

As an example, when the user wants to register a new vehicle in his account, he accesses the *“/new-car”* 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. In this case, it is necessary because only users with an account logged in can register a new vehicle. The other decorator marks the function as handler for the specified endpoint and specifies the HTTP GET and POST requests.

The POST request is submitted by the user and contains the information he has filled in the web form. The algorithm implemented by this function retrieves the data submitted by the user and accesses is using the Flask *request* object, specifically using the *form* attribute. Storing the of *request.form* is done in a dictionary type variable. The data is then accessible using classic dictionary indexing (request.form.get[‘item’]). Using this data, the algorithms create a Car object which is appended to the database, in the corresponding table.

In case the user makes a GET request, the function uses the *render\_template()* function offered by Flask library. This behaviour is expected when the user enters the specific URL and generates dynamic HTML content to the web browser. This operation is standard for Flask's templating engine which integrates dynamic data into HTML templates.

The *render\_template()* function also takes a few arguments. The first argument is mandatory and refers to the name of the HTML file requested by the user. Specifically, in the case of registering a new car, the user expects the rendering of the designated page, which is *"new-car.html"*. This file is stored within the *templates* directory of the application and is one of the pages which construct the interface of the web application. In subsequence, 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 case oif this application, the *render\_teamplate* 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 cars, or other relevant information stored inside the application’s database. This facilitates the generation of personalized content based on the avilable information about the authenticated user context. In other cases, the *render\_temaplate* also takes variables or data that have been previously calculated in the processing algorithms of the route handler.

For better visualisation, the Python code snippet referenced above is Code 1 from the first Appendix chapter. It displays the template of a Flask route handler, as used in the application described by this document. This mechanism facilitates user interaction with the web application, improving to the functionality and use cases of the system. In essence, the short code snippet demonstrates approach of how the web application use Flask’s standard routing mechanism. This mention is necessary as it represents one fundamental aspect of any backend system.

### 3.2.3 Database

At the core of the web application subjected by this document is the relational database model displayed in figure 1. As mentioned before, it is typical for Flask applications to use libraries for Object-Relational Mapping methods to interact with the database. Here SQLAlchemy is used for the creation and management of the database. This library allows developers to define database tables using Python classes. As conventional in Flask applications, the classes are created and managed in the *models.py* file.

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

*ZoneTemporaryData* table is related to Zone table through a foreign key. Specifically, This 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 Low Emission Zones. This is especially relevant for countries like 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 from an instance of *Zone* table using the *temporary\_data* attribute.

Another essential part of the database model is the User table and the ones related to it. The User table is created to store the essential information for account creation. The amount of information is minimal, as the nature of the application does not require more, but this is also favourable for safety and resource storage reasons. The main fields, email and password are need 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 is made using the SHA256 algorithm, which is one of the most secure and used hashing algorithms today [31]. This feature is provided in Python by importing the *generate\_password\_hash()* and *check\_password\_hash()* functions from *werkzeug.security* library [32]. 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 save his vehicles and his 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 with his id. While in reality a vehicle has plenty of details, the application is designed to store only the data relevant to validating the access in the already existing LEZs. The completion of this data is the user’s responsibility and he should respect the information registered in the vehicle book.

Routes are saved in a separate table (*SavedRoute*), which is also related to the User table through the foreign key. Again, the one-to-many relationship implies that a user is able to save multiple routes. This comes as a huge help for frequent travellers and larger enterprises. As for now, the maximum number of destinations in a route is not limited, creating a database table for this model has one clear solution. The destinations are saved in JSON format and saved in *destinations\_json* field, which is used in the application logic, but 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 extends the abstract class *GeneralRegistrations.* This means that for each country a table is created for storing the available registrations or LEZ entry criteria. The OOP principles allows the creation of the abstract class, serving as the superclass (parent class) in this case. This can be extended by other classes, referred as children or subclasses. The inheritance properties allow the subclasses to access, use or redefine the parameters and the methods from the superclass. In this case, the *GeneralRegistration* is an abstract class and serves as a blueprint for the tables storing country specific registrations. Being an abstract class, it is not mapped to an actual table in the database. The superclass defines the relevant attributes and implements the class method *find\_best\_registration\_badge*, which 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 but also based on the last registration date of the vehicle. This adjustment was needed in order to comply with the ongoing regulations in Poland. Based on the values stored in these fields, the *find\_best\_registration\_badge* also takes a different form each country.

Figure 1 represents the database model for this application. It defines the structure and relationship between the elements of the database. This representation was created using the UML class diagram. The building blocks for this diagram are classes, which are mapped to tables in the SQLite database.

A computer screen shot of a computer

Description automatically generated

Figure 1 Database model

### 3.2.4 Access validation algorithms

The architecture of the system encompasses a different algorithm for each country that has a LEZ schema. This strategy is necessary for covering all possible scenarios, intended to ensure the validation of the relevant criteria. The structure is designed to ensure easy implementation for future upgrades. System scalability is mandatory due to the evolving nature of the Low Emission Zones. In order for the application to stay relevant in the future, 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 objects 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, therefore deploying updates will become much simpler. While the responsible researchers can monitor the evolution of each country separately, 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 in order to display a short summary of the relevant information. The displayed information includes a concise statement regarding the access in the selected LEZs and other key details, facilitating user compliance to the regulations. This way, the 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. By pressing the button GO, the POST request is sent to the application backend and the relevant algorithms are triggered. For each country listed in the request, the corresponding algorithm is triggered. Since most countries have different restrictions from one region to another, the algorithms are designed to check for the zone 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 highly challenging when checking the access for several points of interest in the same HTTP request, as it involves structuring a response based on the verification of different vehicle attributes.

While the frontend interface of the Route Planner page seems simple, the database queries, the POST request and the validation algorithms are all processed behind the scenes, in the application’s backend. As soon as the POST request is received, the navigation function handles the data inside the request. 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.

The type and the information contained by the notifications is decided based on the values returned by the access validation algorithms. Notifications can be of type success or error. The type attribute is relevant in creating a suggestive design of the notification. The text of the notifications is queried from the Zone database through a series of class methods. These methods are created inside *models.py* file and take the city name parameter. Each of them is responsible for retrieving key information for the user scenario. This includes information like short description of the LEZ, penalties for not complying, official authority’s webpage, etc. All this data is stored then in the JSON format and sent as a response to the frontend of the application. There, the JavaScript layer of the system unpacks the information. 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.

### 3.2.5 Dynamically rendered web pages

Complex web applications include integration of multiple HTML pages. In some cases, this work can become redundant. Fortunately, modern technologies include tools for creating dynamically rendered webpages. This means that the backend layer of the system configures the data to be displayed in the interface. Generally, this is processed by set of predefined functions and algorithms. These methods also allow the configuration of dynamic URL end points. This means, that the system can have one route handler for several HTML files that are similar between them. The response returned by the route handler depends on the URL accessed. Overall, this technique reduces the amount of redundant code and consolidate the architecture of the system, allowing the control from the backend engine.

Creative software system architectures also take advantage of the database model in order to render the web elements. This removes the need to manually create all the elements of the HTML files. Specifically to this application, the *Zone* table stores important data about the Low Emission Zones which is used in the algorithms described above. Furthermore, this table is used to dynamically render part of the information pages for each country. For implementation, the route handler receives a parameter which represents the country selected by the user. This selection takes place in the graphic interface and can be done in several ways. Based on this attribute, the route handler makes database queries, processes the data and structures the response. To render the generated HTML file, *render\_template()* function is used again, only this time also takes the parameters previously calculated in the route handler. One of the dynamic elements created in the DOM of these country information pages is the div that contains the LEZs. Basically, this is an unordered list, created by the <ul> tag and contains 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, to achieve the same result each list would need to be created manually. This is not only redundant as there are several countries and the zones are already saved in the database, but using dynamic rendering, it is sufficient to implement the updates in the database in order to have them displayed to the user (no additional modification is needed in the HTML files).

## 3.3 Frontend Development

### 3.3.1 General aspects

Frontend Layer represents the graphic interface used by the client to operate the system. In the case of web applications, this interface is rendered by the user’s web browser. A good design should take into consideration the looks of the application but should also arrange the elements in a suggestive or intuitive way. This is important as most users or stakeholders do not have technical background. Complex applications use the frontend layer to abstract away the different complexities of the applications and hide them for the user. Depends on the domain of the application, but generally the experience should be simple and enjoyable. Modern applications set high standards for user interfaces as the frontend interfaces are expected to be responsive enough to ensure real time communication between the user and the backend.

Essentially, 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 like JavaScript, the design of such pages is plain and their functionality is static, limited to predefined text. From an architecture perspective, HTML templates can be viewed as the building blocks of the application, using the tag elements as bricks, while CSS adds the design and JavaScript sets the functionality and interconnects the elements. Typically, Flask application store these HTML files inside the *templates* folder and CSS and JavaScript files are stored inside the *static* folder. 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 important in web applications, which often use a combination of different programming languages and file types. This practice is rather mandatory in complex systems as it also creates order and well organization for the developers.

HTML files contain the head element (*<head*>) where they link static files. This linking allows for the 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. These can be used by identifiers, both in the stylesheets (CSS files) or in the scripts contained in the JS files. While the ID should be unique and refers to a particular element, the CLASS attribute refers to a bunch of items which have the same design or behaviour. Importantly, one HTML element can have several CLASS attributes, making the design process much complex, as the classes can contribute to create elements. Also, this makes accessing the DOM in JavaScript much more feasible.

One of the advantages that comes with the use of Python, specifically Flask for backend, is the possibility to use the available templating engines. While the aspect of rendering dynamic web pages, using the pre-processed backend data was described in chapter 3.2.5, 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 one web application have the same design theme, meaning they look and feel similar because they often reuse some elements. In Flask, this can be easily achieved through the use of Jinja templates. The web pages of this application are created as an extension of *base.html* file. In Jinja, the sentence for this is simple yet extremely effective. Creating the base template for the file should include the *{% block %}* elements. Inside of these blocks can be used classic HTML elements, to create the structure of the template. Then, in the rest of the files, the extension is done 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 the buttons for accessing the most important resources and functionalities of the application and is present on the top of all webpages. Here, in the application subjected by this document, the navbar element was created in the *base.html* file. We need to remember that this *base.html* is not accessible to the user and it is just a template. In fact, all these other pages are an extension of this *base.html* file. This restriction is not directly implemented but derives from the fact that no user accessible endpoint is defined to 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 some elements, that are used on different pages can be defined in their specific files and only be linked once to the base file.

### 3.3.2 Maps API

While highlighting the different LEZ can be done through different approaches, the application uses dynamically rendered maps created using the Maps API. The large catalogue of Google services includes different options, out of which Maps JavaScript API 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.

Firstly, the application displays all the available Low Emission Zones right on the Home page. Maps API allows the creation and placement of markers representing specific locations. Here, these markers are automatically created at runtime, based on the Low Emission Zones queried from the database. The backend of the system is responsible for the query when the user sends a GET request. Then, the creation of the map element and the population with the corresponding markers takes place in the frontend of the application. JavaScript functions parse the response, which is a JSON format element containing countries and cities. In order to create a marker from the name of a location, the address must be put through the geocoding process. This process is completed by another API offered by OpenCage. The maps are customized using the attributes defined by the API. This application slightly adjusts the default behaviour of the map element and allows users to roam freely. Specifically, the user 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. For better identification purposes, a dynamic list is joined next to the map. Both elements were created with the intend of easily highlighting the LEZs.

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. The route is calculated by the Directions API from google, therefore offering best results. 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 which is responsible for handling the form submission. When the user clicks the *GO* button, the web browser submits a POST request containing the data of the HTML form. At 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 a conventional address, not geographical coordinates. This time the geocoding process is done directly using the Google Geocoding API, not though an additional third party.

The choice of splitting the geocoding process between two service providers has been 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 hard cap 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.

### 3.3.3 OpenCage Geocoding API

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, making it even more suitable for the nature of this application. This is necessary because the application includes a large number of Low Emission Zones in the database and marks a waypoint on the map for each of them.

The geocoding technique represents the conversion of a location or an address to geographical coordinates (or reverse) expressed in latitude and longitude [44]. These coordinates are the input 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, the geocoding would still be necessary in the Route Planner page, as the possibility of addresses is basically infinite, as the user could select not only cities, but also full addresses.

### 3.3.4 Air Quality API

Considering the fact that Low Emission Zones intend to reduce air pollution in major cities, one of the requirements set for the application is to inform users of the air quality in different European cities, hoping it would considerably improve the LEZs efficiency. Fortunately, air quality is monitored and scored by the World Air Quality Index (AQI) project [45]. 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, a good air quality is considered when the AQI is below 50 and still acceptable for AQI under 100.

The API offered by the [45] project allows developers to integrate AQI data in their application. They can take different approaches, like creating widgets or displaying markers on maps generated by other API providers. This application takes the second approach and uses the combination of the 3 APIs to render markers which reflect the existing LEZs and the AQI in real time. Over more, the air quality is taken into consideration by the validation algorithms for Budapest LEZ (LEZ is only active when AQI is above 50) and could be extended to other areas if necessary. Anyways, the notifications displayed on the Route Planner page of the application also include the real time AQI.

A map of europe with red points

Description automatically generated

Figure 2 LEZ Map - Home Page

### 3.3.3 Asynchronous function

Asynchronous programming is a key feature of JavaScript and is usually done by creating special asynchronous functions. These functions use the await key word structures 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.

The system creates asynchronous behaviour on the Route Planner page. Pressing the *Calculate Directions* button, the user triggers an asynchronous function. The frontend layer gathers the user-filled information and saved 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 user-filled information (destination address) that was previously saved in JSON variable. This object is included in the body of the POST request that will be sent to the application’s backend. Afterwards, the frontend interface suspends execution until a response from the backend is received. Basically, the process is paused (using the *await* key word) and creates time for the backend layer to receive the data, process it and send the response. Frontend interface waits for the backend algorithms to run and send the results.

This creates a seamless interaction between the user and the system, as the response is received in real time. This technique allows DOM manipulation and real-time update of the interface without the need to refresh.

### 3.3.4 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 also tied to the asynchronous event described in the previous subchapter. Shortly these notifications represent the results of the access validation algorithms, which are transferred to the frontend by the response of the POST HTTP request sent by the async function described before.

For clarification, the text and the type of the notifications are generated by the access validation algorithms, which interact with the database as described in the backend chapter. Based on this text, the notifications are created dynamically by the JavaScript algorithms. Essentially, they create <div> elements which are then appended inside the HTML page. For these elements the algorithms set the values for the class and id attributes, based on the notification type received from the backend algorithms. The goal is to not overwhelm the user with information, but to inform him only with the information relevant to his specific case. These attributes are configured in the stylesheets to offer a suggestive aspect the notifications. For example, if the access validation algorithms decide that the selected vehicle is not eligible for access inside the LEZ, a red alert is raised. Color-coding the alerts and notifications is a common approach as red or green colours already have a signification encoded in the population’s mind. Based on the classes set by the JavaScript functions, the notifications are styled in the already linked CSS files. There, the different different attributes for aspect and positioning are defined. This way, when appending the elements to the DOM, the frontend system already knows how they should be rendered. Since the number of notifications is dictated by the number of detitanations, the flex-box approach is used to fit all notifications in a fancy looking and easy to read manner.

All of the 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. Over more, the user is directly guided to check if his vehicle is eligible to obtain the required registration.

Each address selected by the user generates a separate notification to summarise the situation.

A screenshot of a car registration

Description automatically generatedA screenshot of a computer

Description automatically generated

Figure 3 Access Granted Notification Figure 4 Access Forbidden Notification

### 3.3.5 Dynamic elements

This section describes how user interaction affects the structure of the web pages. This refers to information 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 and HTML form. Rather than asking users to complete a long form 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.

For new users with minimal knowledge of Low Emission Zones (LEZs), 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 Spanish Emission Stickers. This way, the process of selecting the appropriate registration for their vehicle is made more intuitive.

Additionally, the notifications showcased in the previous chapter are also dynamic HTML elements. Their presence is dictated only once by user action. As outlined, the information and the type of these notifications are dynamically generated. Shortly, they present relevant information without suffocating users with long text paragraphs.

Furthermore, when selecting destinations in some specific countries, such as Italy, users are prompted to input the period of the trip separately. This information is very significant, particularly because of the different LEZs regulations during the winter period and is factored into the validation algorithms. However, for most countries, this requirement does not apply and therefore soliciting this information would be meaningless.

These dynamic fields are typically accompanied by remove or delete buttons to facilitate easy correction in case of user errors or updates over time. This functionality allows users to quickly rectify mistakes or update information as needed, ensuring a smooth and user-friendly experience.

This approach aims to create an interactive form that only necessitates completion of select fields tailored to each user, thereby enhancing efficiency in user interaction while minimizing time consumption. The selective retrieval of relevant data also greatly benefits the efficiency of the algorithms and system, reducing both processing time and resource utilization.

To achieve the described behaviour, the application uses custom JavaScript functions and event-listeners to add and remove HTML elements. These functions utilize the Document Object Model (DOM) to create new elements, set their attributes, and append them to predefined containers within the HTML structure.

Additionally, there are similar functions for removing the elements if necessary. These removal functions traverse the DOM to locate and delete the targeted elements, ensuring efficient management of dynamic content on the page.

### 3.3.6 Conditional selection

While granting users the freedom to navigate the application is favourable, guidance in certain selections ensures prevention of errors. This precaution is crucial for validation algorithms to receive accurate input. Given the wide variety of Low Emission Zones, the access validation algorithms encompass a multitude of scenarios, therefore it is mandatory for them to receive accurate input.

In addition to users receiving accurate responses, thus aiding in regulatory compliance and avoiding potential fines, guiding users through certain selections also enhances their overall experience. The frontend interface helps users in data selection. Ensuring smooth user-experience increases the efficiency of the time users spend on the application.

One situation where users are guided in their selection is when registering a new vehicle. Here, users are prompted to input the brand and model for identification purposes. Subsequently, after selecting the brand, the model input field is automatically populated with relevant options corresponding to the chosen brand, simplifying the selection process for users. While technically this does not improve the system functionality, it improves the user experience, which is an essential requirement.

Similarly, in the vehicle registration process, the user needs to select fuel type (petrol/diesel/hybrid/etc.) and Euro Standard. In order to improve user experience but also avoid logical error, the interface conditions the selection again. For example, users are unable to choose a zero-emission Euro standard unless the fuel type of the vehicle is electric. Conversely, an electric vehicle can only be registered as zero-emission. While users are responsible for verifying the vehicle specifications in the registration documents in order to accurately complete the form, these restrictions help to mitigate potential errors. Implementing these constraints helps prevent illogical scenarios, such as registering a Euro 2 Electric car, which not only lacks practicality but also risks raising errors during the validation process. By enforcing these restrictions, the system promotes data accuracy and reliability, enhancing overall user experience and regulatory compliance.

Another notable aspect of how the interface guides user interaction is on the Route Planner page. Here users select points of interest using a well-known autocomplete functionality. The suggestion options are generated by Google Places API. This feature stands out as one of the most widely used autocomplete tool, as it is specially designed and configured for location prediction. This significantly enhances efficiency, as frequented places are suggested after just a few letters are typed in the box. This software is extremely versatile, as it includes various points of interest, from city selection that usually reflects city centre, to full addresses that include street names or even numbers, hotels, cultural monuments, or any other points of interest.

Moreover, the use of autocomplete function ensures accuracy of selected places, guaranteeing precise input for the Google Geocoding API. In order for the route and the markers to be set on the map, the geocoding process must convert the locations into geographical coordinates. Any small typing mistake made by the user 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 in the backend chapter, these city and country are considered in the access validation process. Overall, managing the destinations selected by the user significantly reduces user time to input and optimizes functionality, resulting in a smoother user experience.

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

A screenshot of a computer

Description automatically generatedA close-up of a route

Description automatically generated

Figure 5 Autocomplete API Figure 6 Browser Console After Selection

## 3.4 Software Development Moldes and Tools

### 3.4.1 Integrated Development Environment (IDE)

Integrated Development Environments represent 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. Their ideal for extensive programming sessions, as they improve programmer efficiency, offering code completion, refactoring or syntax suggestions based on the selected programming language [34]. 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 is the IDE used in the development of this web application. It 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, usually offering a smoother experience compared to other IDEs. Some the key aspects are the integrated terminal for quick debugging and commands, possibility for extensions like IntelliSense or Git Integration for easy version management, customizable interface or even live server extension for fast web development. For this application, VS Code perfectly integrates Python language, with the possibility to install libraries or frameworks using the pip command directly in the command line.

### 3.4.2 Virtual Environment

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 developing this application is represented by the setup of a new virtual environment. This process was done using a dedicated software.

Named Anaconda, this software application is designed to structure package and environment management, specifically for Python and R programming language [36]. This allows developers to create different virtual environments on the same device. Each environment represents a separate and isolated workspace that encapsulate the prerequisite software required to compile and run the code of the application. This isolation allows developers to effectively integrate and manage the dependencies between the different libraries. Separating the projects to specific virtual environments ensures that the applications benefit of the required version of the libraries and also avoids conflicting packages. This approach is crucial in modern software development, allows easy bug fixing, ensures the global environment of the host does not get corrupt from conflicting libraries but also ensures smooth upgrades.

Anaconda Prompt and Anaconda Navigator are two of the tools which allow the creation and management of 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 allows smooth and optimal setup of the libraries required in the software development process.

### 3.4.3 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 used DB Browser, an application dedicated for management of SQLite database files. This open-source tool disposes a guided user interface for visualisation of the database. Additionally, in here developers can edit the tables, use the search function for quick data visualisation but can also use SQL commands and queries in the dedicated terminal.

During the development of this application, the database tables were created in the *models.py* file, using the Object-Relational Mapping properties of SQLAlchemy library. Additionally, DB Browser was used to populate the fields of the tables using SQL statements. Considering the complexity of the database model, this application was extremely helpful in data visualization, offering the possibility to switch between tables but also to make slight adjustments when necessary. The application allowed easy interaction with the database, benefitting the update implementation.

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 order to facilitate database migration within the application. In essence, these migrations represent changes to the database schema, which only become effective after the commit takes place. Such modifications include adding or removing tables and modifying columns of existing tables. The scripts automatically generate and commit the migrations based on the latest modifications detected in the *models.py* file. Once generated, migration scripts are applied in an incremental approach to update the database schema. Flask-Migrate tracks the current state of the database schema and applies migrations sequentially 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 migrations are saved in the *migrations* directory.

## 3.5 Development Techniques

### 3.5.1 REST APIs

Representational State Transfer Application Programming Interface (REST) API are responsible for the communication between the caller and the provider of a resource. They are frequently used in 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.

The call of a REST API transfers a representation of the resource’s state to the endpoint. The endpoint refers to a specific URL, accessed by the consumer of the resource, with the scope of performing some action or retrieving information. Typically, each endpoint has a particular resource associated. This is retrieved in specific format type, via HTTP requests managed by the client-server architecture. The communication is considered stateless, as each request is independent and the GET requests do not store client information [36]. 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 contain all the necessary information. Important information such as authorizations metadata, uniform resource identifiers (URIs) and are contained in HTTP request headers [27].

This application makes use of the REST APIs to handle POST and GET HTTP requests. The endpoints are defined in the *views.py* file and handle the retrieval of information with the GET request or update the resource using the POST request. The information is transferred between layers of the application, from client to 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.

### 3.5.2 JSON

JSON is the short abbreviation for JavaScript Object Notion and despite the name, this format is independent of the programming language, as it can seamlessly be used in C/C++/C#, Java or Python. The JSON format offers a lightweight method of transferring data, as it is easy to read and interpret by both machines and humans because its structure and elements are familiar to most developers.

Elements of JSON format are structured in two parts. First is a collection of name and value pairs, similar to dictionaries in Python or other programming languages. Secondly an ordered list of values, which is similar to how lists or arrays work in most programming languages [37]. In short, JSON formats data in a list of dictionaries. This format composed by data structures familiar to most programming languages ensure JSON’s versatility, making it one of the most frequently used data formats for transferring information between systems.

The communication between the backend and frontend of this application includes the data transfer using the JSON format. Python offers the dedicated JSON library, which was imported in the project, 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 can contain data which needs to be submitted again to the frontend, in order to be displayed to the user. Again, the data is packed in the JSON format and this time is parsed by the frontend algorithms.

Using JSON format ensures the integrity of the data, compatibility with the two programming languages but also because of its structure, the debugging process was fairly intuitive, as the plain text of this format is readable by humans.

### 3.5.3 CRUD in Web Applications

CRUD operations are fundamental in web applications and are often used in RESTful APIs. Each endpoints provides a method for serving an HTTP request and allows clients to interact with the backend to perform these basic actions. CRUD stands for Create, Read, Update, and Delete. These are the four basic operations that can be performed on a resource of a database. These operations 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.

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. SQLAlchemy makes the process of interacting the database straightforward, ensuring that data management is efficient and organized. These CRUD operations form the essential building blocks for all web applications, enabling the creation, retrieval, updating, and deletion of data in a structured manner.

### 3.5.4 AJAX

AJAX represents a set of techniques used in web development and is short for Asynchronous JavaScript and XML. This ensures the client-server communication takes place without the need to reload the page after every HTTP request. It is made possible by structuring data transfer, as the client and the server only exchange small packages of data at a time. This concept improves the overall feel of the application, minimizing interruptions as it allows only partial update of the HTML page. AJAX composes techniques like XHMTL and CSS, Document Object Model, JavaScript or XMLHttpRequests. These make it 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.

Traditionally, in web applications the user interaction is locked during the transporting process of the HTTP request. AJAX sends and receives data from the server asynchronously creating the possibility of handling the server communication without intervention form the client. This means that now users do not need to wait for the HTTP request to be completed and the interface can be updated dynamically without the need to fully reload the web page. This creates the possibility to have real-time user interaction. As an example, the click of a button or selection of an item in the input field will generate an immediate response with a visible update.

The implementation of AJAX in the application presented by this document improves user experience, offering a fluid experience across the web pages, allowing users to test different scenarios of navigation without the need to refresh the website.

### 3.5.5 Document Object Model

The structure of documents can be represented as logical trees, consisting of nodes and objects using the Document Object Model (DOM). In this technique nodes contain objects which have event handlers attached. The nodes are interconnected by the branches of the tree. Using the DOM methods, the tree elements can be accessed by software engineers in order to change the content or the structure of the document [39].

DOM offers different types of nodes for documents, elements, text, attributes, or comments. Objects, on the other hand are instances of classes which define the structure and behaviour of the document. Each node is represented by HTML elements. In JavaScript, these objects can be accessed by methods such as *getElementByID()* and manipulated by methods such as *appendChild(),* *setAttribute()*.

By integrating the DOM and AJAX using 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 friendly interface allows users to directly input data in the DOM elements, which are sent to the backend of the application in an asynchronous communication. 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, without the need to refresh the page.

### 3.5.6 ORM

While the 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 Object-Relational Mapping (ORM) techniques in the management of the database systems [40]. One of the most popular ORM libraries is SQL Alchemy. The ORM techniques create an abstraction layer that maps Object-Oriented elements used in the programming language to relational database elements [41]. 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 instead of executing SQL queries.

This approach puts the focus on data processing and business logic, concentrating the code required to interact with the database in the backend level. Classes mapped to tables can be easily adapted to fit future updates, guaranteeing easy maintenance of the database schema.

SQL Alchemy is a very popular ORM library and is often used in the management of SQLite databases in Flask applications. The application described in this document uses this combination for the database interaction as the tables are created by classes configured in the *models.py* file. The CRUD operations are handled in the endpoint handler functions, which interact with the database using Python operations like *db.session.add()* or *Zone.query.filter\_by*. ORM does not restrict the use of classic SQL syntax. Once the database file has been created, SQLite operations can still be executed. While the tables were created by classes, they were populated using SQL commands (process described in Chapter 3.4.3).

### 3.5.7 Authentication and Authorization

Creating an account allows the user to personalize his experience and interaction with the application as it facilitates the access to gathering and storing information to 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. The authorization determines the actions available to the user, usually based by his role. The authorization aspect is critical for the security of the entire system.

The authentication process typically involves verifying user’s credentials against the ones already existing in the database. Obviously, the user needs to have an account already created and use the credentials saved in the database to authenticate. 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 route handlers for their endpoints.

Considering the nature of the application, the authorization process takes into account three types of users. First are the regular users, possibly first timers, as these represent clients that do not have an account created or signed in. Their access is limited by the *@login\_required* decorator associated to the route handler functions.

Secondly, are those who have already signed in with an account in order to gain access to the features dedicated to the target audience. This requires they would first complete the signup process on the application registration page. Each user session is handled by application backend and authorization is verified by the *current\_user* object offered by Flask [42].

Lastly, the application authorizes access to a special admin user. This one has been created with the purpose of accessing and manipulating the database from the dedicated application page. The admin user role has been created for stakeholders with little technical background, as they benefit of the user-friendly page to edit the table storing data about Low Emission Zones. This role offers authorized users to make database adjustments, when necessary, without the involvement of the developer. The administrator role is considered to be essential in the industry, as some stakeholders should have the possibility to manage modern application. 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.

### 3.5.8 Templating Engine

Template engines represent tools used to generate dynamic content files based on a static template. In the case of web development, this refers to HTML files. This method allows developers to create a static HTML file, which contains placeholders for data and serves as template for other files. On runtime, or when necessary, the placeholders are filled in with dynamic data, offered by the templating engine. This approach separates the HTML files from the backend of the application, promoting cleaner and maintainable code [43].

Jinja is a powerful templating tool for Python, and it is possible to integrate with several web frameworks but is best used in combination with Flask, as it uses Jinja as 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 very familiar to Python developers, as it’s been derived from it. Jinja offers the possibility of inheritance, which was also used in the development of the application. 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. In Jinja this includes CSS or JavaScript files linked in the head of the file or even entire HTML elements, which heavily reduces the amount of code.

As explained in chapter 4.3.1 this application makes use of this technique, in order to favour code reusability and create the theme of the website. The application also uses *render\_template()* in order to display the HTML page to the user. This means the files are rendered on server side, as the HTML files are fully generated before being sent to the client. For some end points, the application combines template rendering with REST APIs for the CRUD operations, as this involves that the client consumes the API, and based on the received data, updates or generates the user interface.

# PART II – PROJECT CONTRIBUTIONS

# 4 Project Motivation

## 4.1 Addressing the Complexity of Navigation throughout LEZs

Low Emission Zones are not only beneficial, but essential to the well-being of our society and future generations. Efforts must be paid in order to reduce pollution, as the effects of global warming cannot be neglected nor denied. As time passes, more and more European countries adopt legal frameworks to regulate the access of the highly polluting vehicles. In the past few years, hundreds of cities have adopted different variations of Low Emission Zones.

The wide-spread presence of the Low Emission Zones is a good step towards mitigating pollution in the urban areas. The problem is caused by the differences between the legislations imposed in different countries. Some people may not be affected by this if they do not travel often or even so if they do not use personal vehicles. However, many citizens are confused by the different restrictions they must comply to. The issue in itself is not with the number of Low Emission Zones, but with the fact that each Low Emission Zone is different from another. Confusion raises as in most countries the access requirements differ from city to city. We must acknowledge that this is still an emerging topic, which is going to evolve over time and presumably a definitive schema will be adopted throughout all Europe. But until then regulatory measures must be taken.

What is more concerning is that available resources often fall short in providing clear details about the different restrictions and the measures that must be taken to comply. While online resources and information are indeed available, usually the information 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 their 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 for 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, the navigation throughout the European Low Emission Zone is more difficult than ever before. For vehicle restrictions to be effective, population and official authorities must comply. Unfortunately, in many cases, if the information is not accessible or concise enough for the regular individuals, the restrictions imposed by Low Emission Zone are violated. In this case, even if authorities impose further penalties, the scope of Low Emission Zones has failed, because the pollution is still on the raise.

## 4.2 The Recommended Solution

This project composes a solution for reducing the chaos and confusion created by the different Low Emission Zone schemas. This it the development of a new web-based application that serves as centralised hub for information on European Low Emission Zones. 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 regular individuals, who drive personal vehicles in countries that embraced Low Emission Zone frameworks. The goal of creating this application is to eliminate all barriers of entry, such as subscription fees or prior expertise in the domain. By doing so, the application empowers a wider audience to make informed decisions before buying a new vehicle, while also expanding the impact and reach of LEZs.

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

At the core, the application makes use of a robust database and extensive algorithms for assigning the correct registration to a vehicle and LEZ access validation. These technical features 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 app keeps the same high standards for information quality, consistent with the other applications already established on the market. The purpose is to deliver up-to-date, accurate information. In addition, the application contributes to spreading awareness about Low Emission Zones and promoting the social acceptance of urban vehicle regulations. Users are encouraged to take smart and responsible environmental decisions, essentially taking a significant step in boosting the effectiveness Low Emission Zones. Through its unique manner, the application ultimately contributes to the larger scope of improving air quality in European cities.

The interface of the application must be centred around the user. This involves guided interface which conduct the user interaction while also 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 main method of interacting with the application is through the smart algorithms. They are encapsulated behind an intuitive interface which allows users to quickly obtain the information they require.

# 5 Development Methodology

## 5.1 Analysing Requirements and Specification

The development of this web application was structured in different stages, following the principles explained in [33].

The first step in the development lifecycle of the web application presented by this document is to define the project theme. This involves the establishment of its based on global requirements and constrains of implementation. Considering this, the application represents a necessary tool, which facilitates the access and compliance to Low Emission Zone regulations. The platform for deployment is the internet, as the application must be reachable to the majority of population. This offers the possibility to integrate modern technologies in order to encompass a universal hub, where people can get the necessary information. Already existing resources are text based and are not accessible to everyone, either because the information is not concise enough or because they have associated use-cost.

Based on the specifications set in the first step, the modelling of the overall design of the software application can start. This includes setting the requirements and constraints of the application. In this case, the software system needs to include the modelling of an extensive database, which comprises information for vehicle restrictions in urban areas and system users. Also, the system needs to facilitate information access through special algorithms that determine whether specific vehicle is eligible or not to access points of interest. For this reason, the system needs to permit user account creation and registration of personal vehicles. Notable constraints are that for the start the application should focus on Low Emission Zones and the type of vehicle which is the most used.

The detailed design continues to set the requirements specified in the previous step of the development life cycle. It involves gathering information, documenting and understanding the requirements and expectations of the system to be developed. So far, only the basic structure of the application has been designed. This includes deep details of the requirements, like deciding on what technologies are used, technical configuration of the application routing, delimiting the countries and cities covered by the application database, selecting information relevant for user account creation. Also, in this part, the overall look of the application and the content of the web pages was set. While the server-client model was an easy pick for the architecture, due to its nature being perfectly suitable for the web applications, this part also involved decisions regarding the basic functionality of the validation and the eligibility algorithms. For more complex pages, like the Route Planner, the planification also includes the planning the sequence of actions, starting from user input all the way to displaying the response generated from backend.

Further, the development process is based on the analysis of the requirements and technical aspects described above. The main topics composed the setup of the application backend routes, algorithm implementation and refinement and database model implementation. All of these topics were described in chapter 3. The frontend elements of the website were created afterwards when the API routes were already functional at a basic level. Taking this approach, allowed focus on application functionality and comprise the interface to fit it, as the dynamic design is based on user interaction and backend response.

### 5.1.1 Information Sources

The research process of designing and developing 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 refers to the relatively new and innovative topic of Low Emission Zones, traditional books are generally rare or non-existent. Nevertheless, online resources represent other credible repositories providing valuable information on this topic. The analysis and development of the application considered the types of resources outline below.

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

On the other hand, academic research or conference papers often offer information regarding the newest case studies and their findings. While momentarily the focus of this application is not to analyse and select the more effective Low Emission Zone framework, having an overall understanding of the topic helps in structuring and delivering the information to the user. Other studies may reflect the future of the topic, for example the study by Statista.com, which highlights the abundance of Low Emissions Zones, based on which the decision to create a scalable solution for the database model.

As part of the analysis process, reviewing already existing applications provides practical insights into real-world examples of how the system can be helpful for the user. From a research perspective, we can see how the information is delivered and compare this information across other sources, achieving often contradictory results. While the clarification for this issue was discussed in the second paragraph of this chapter (to opt for official websites that are up to date and accurate), the analysis of existing applications, offer additional benefits. From a technical point of view, these examples can reflect challenges faced by these applications and their solutions, but also highlight technical topics where other applications could improve on. Overall user experience, application time/resource efficiency or available features can also be analysed from a technical point of view, as they are mainly part of the application backend. Additionally, this offers a business perspective, highlighting the places where the new application can improve on.

The resources described so far represent the foundation needed for the analysis part of the development lifecycle. Based on these the choice of the used technologies was made. While the technological features and how they were used in the project were described in detail in the third chapter, the choice for these technologies came after long research. This relies on a variety of resources, from traditional books on programming, to online documentation for frameworks and libraries or community support offered through trusted websites.

For the frameworks, libraries and APIs used in the web development process, like Flask or Google API, the main resource is the official documentation page. Here, developers can find the latest updates, code examples, features compatibility and work patterns but also frequent bug fixes or questions to FAQ (frequently asked questions). The database structure was modelled based on the principles and practices of relational database, described in multiple documentation and development tools. Community support, in general offers reasonable solutions to frequent problems, innovation or refinement idea but is also generally more exposed to possible bugs and necessitate full understanding of the entire application in order to include features from such sources.

### 5.1.2 Application Use Cases

Once the analysis of the project requirements is complete, the next step is to design the overall schema of the system. This process is effectively structured and visualized by the Use Case Diagram. As part of the UML, the Use Case Diagram provides a visual representation of the interactions between users and the system.

In UML, the use cases are represented by an oval and should be named suggestively for those who read the diagram. The use cases represent possible scenarios of the communication between the system and external users. As part of the diagram, these external users interacting with the system are represented by a human figure and they are referenced as actors. The possible scenarios of interaction between actors and system are represented by use cases, which represent a sequence of actions. The purpose of this sequence is to provide the user the intended action. 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 or extension. Inclusion means that the execution of one use case implies the execution of another case. Exclusion means that one use case is conditionally triggered by another.

The interaction between the users and the web system descried by this document was represented in the Use Case Diagram, Figure 7. 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 on the left represents regular users, who can initially access the home page and the pages designated for each country’s LEZ schema. Here, they can find general information about the application and how it works but also text-based information about the different Low Emission Zones. In order to gain full access to the features of the application, users must create accounts and save their vehicle. 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 are able to access the edit vehicle use case, but of course that is only possible for already saved vehicles. The second actor in this diagram, from the right side, represents the administrator of the application, who has a special account type, which allows him to edit the database of the application from the web application. The admin user, also referred as technical user, is also able to use the other functions of the applications, but the visual of representation for that would be unnecessary.

A diagram of a network

Description automatically generated

Figure 7 Use Case Diagram

### 5.1.3 Access validation process

The activity diagrams are also part of UML standard and represent the flow of control or data within the system. They are used in the modelling of dynamic aspects of a system and generally illustrate the sequence of activities triggered automatically. Activities represent steps in the execution of an algorithm and only have entry and exit transition associated. These transitions, are generally called flow controls, represented by arrows and are triggered automatically by the end of an activity. Additionally, in the activity diagram, developers use 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 add complexity in the diagram is to use concurrency elements when representing synchronization between parallel activities.

A diagram of a company

Description automatically generated Figure 8 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 represents the asynchronous behaviour of the application, as the frontend interfaces interrupts the flow and waits for backend response. Afterwards, the notifications are displayed, which means user has received all the information he wanted, and the sequence is finished.

Figure 8 Activity Diagram

### 5.1.4 Non-functional Requirements

The functional requirements refer to the capabilities and features application and how it interacts with the users. Shortly they can be referred viewed as utility aspects of the system. These requirements were described above, and the interaction capabilities were depicted in the Use-Case diagram. Non-functional 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. The non-functional requirements of the application were defined in the system analysis process.

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

Real time response to user interactions should be ensured by the system, maintaining the performance even during high load conditions.

Security and integrity of the data is important. User data should be encrypted and the system should be protected against unauthorized access to avoid data corruption.

### 5.1.5 Application Constraints

For the start, this application has been developed to cover a specific niche in the market, therefore its utility is limited to meet the requirements of those individuals. Nevertheless, the effort put in the analysis of the urban regulations subject and in the development of the web application, serves as a fundamental step in creating a large-scale application, meant to become a central hub for traffic restrictions and regulations throughout Europe.

First of all, the application is limited to a single type of vehicle, specifically passenger cars. The choice to fit in these vehicles was made because they are the most widely used for personal transportation and are the ones most affected by the newly European regulations, because they are typically powered by combustion engines. Since the Low Emission Zones and other traffic regulations impose different access criteria for each vehicle type, a viable validation system needs to also consider this aspect.

The second constraint of the web application is represented by the inclusion of only Low Emission Zones. While the urban regulations impose different strategies for reducing pollution, these ones are affecting the majority of population [1]. This results from the nature of the restrictions as the target represents the central areas in large cities but also from the wide spread of the zones. The application addresses Low Emission Zones because they are the most effective form of pollution reduction in central areas and because the modelling of the different regulations and the development of a universal access validation system is a challenging aspect, which at the moment is still to be discussed.

## 5.2 Design elements

### 5.2.1 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 clients which make the requests to the server. Generally, the communication takes place via the internet and allows multiple users/clients to connect to the Server at the same time. The Client-Server model 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 make requests to the application. The Server is basically the web application, which is running on the local machine and started using the built-in Flask development mode. This is ideal for applications testing 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, described in the third chapter, receives the requests and routes them to the appropriate handler functions. These functions are responsible for handling user information, whether it is a form that was submitted or 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 and maintainable and improves efficiency by centralizing processing tasks on the server. Thus, the Client-Server model used in this application, 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.

A screenshot of a computer

Description automatically generated

Figure 9 System Architecture

# 6 Description of the Practical Application

## 6.1 User Guide

The web application was designed to be utilized by any end-user, even by those with no technical background. The features of the application and the of using it are described at the top of the home page, so the user can see them when he first enters the website. Over more, the home page offers general information on Low Emission Zones, both in text and image format.

As described in this document, but 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 for the scope of this application and would only complicate the process of creating an account. Right after the account creation, the user is logged in automatically and the home page is refreshed in order to render the newly unlocked features. The web navigation bar, situated at the top of every web page, allows user to easily navigate across the pages.

Next, the user should register his vehicle. This can be accessed either from the home page, by clicking the suggestive image, or by accessing the My Profile page. There, the user can manage his vehicles, add or edit the specifications, but also edit the routes saved as favourites. The vehicle registration process is completed on a different web page, in order to shift the focus on the details filled in by the user. Here, the application offers an intuitive form, where the user is guided to fill in the relevant details. Brand and model are necessary for identification purposes, while the rest of the details are relevant for storing the description of the vehicle, which is necessary in the Low Emission Zone access validation process. On this page, the user can and should add any registrations that are already assigned to his car. This possibility is offered by the dynamic form element, which guides the user in the selection process. Any mistakes made or feature modifications can easily be adjusted even after the car was saved. The application allows users to edit the specifications of their vehicles and uses intuitive symbols for adding or removing elements. Once at least one vehicle is saved, the user can fully benefit of the features intended by this application.

The Route Planner page can be accessed from the navigation bar or from the Home page. Here the user should fill the form, pick his vehicle and select the destinations with the help of Autocomplete. The number of destinations is completely adjustable. By default, the page loads one field for destination, but the user can add several, by the click of a button. These additional fields can also be removed if necessary. For these actions, the user is guided again 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. The notifications suggest that he can also use the Eligibility Check page, to verify his vehicle for potential registrations or compliance with National LEZ schemas.

Eligibility page can be accessed either from the navigation bar or by clicking the link offered by the notifications received on the Route Planer. Here the user has the possibility to select one of the vehicles saved to his account or to fill in the relevant specifications of a different vehicle. This can be ideal for users who are considering changing their vehicle or buying a new one. Thus, they can compare the situation and get a better understanding of the regulations, resulting in a better-informed acquisition. The system only checks the eligibility of the second option (new vehicle specifications) if the field for vehicle selection is left unchanged. Lastly the user should select for which country he wants to test. After selection and the click of the lens-icon button, the user is instantly informed of the registration eligible for his car. This also includes an image of the application (where it is applicable), general information of the National LEZ schema and the link of the official regulatory website.

This process lets the user know if he needs any registrations for his trip, if his vehicle is eligible to obtain them and where they can be acquisitioned from. The system takes into consideration the different possibilities, of whether or not the selected destinations impose a LEZ and vehicles compliance to the schemas. Once the user gets a new registration for one of his vehicles, he should also assign it in the application in order to ensure the correct information for the validation algorithms.

## 6.2 Application Testing and Validation

The testing phase is vital in the development lifecycle as the process ensures that the application system works as designed and meets the requirements and specification set during the analysis phase. This can be achieved through different techniques and methodologies, depending on the nature of the system. Vigorously testing any application should be both qualitive, to ensure that a system functionality behaves as designed, but also quantitative, to cover the different scenarios.

Unit testing is a viable solution for testing the API endpoints and basic functionalities, like access and authorization for users and data transfer. Python offers the pytest library, which ensures the possibility to write and run unit tests on the Flask route handlers. The principle applied in the testing of this application was to write testing scripts with predefined input and expected output. The scripts run the different end points 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), in order to ensure smooth operability and avoid data faulting. The planning and execution of the test units covered the validation of user access and authorization but also the access validation algorithms. The main advantage of using test units is the saving of time resource, as once the test unit scripts are written, the actual testing can be started from the console offered by the IDE and takes seconds.

Integration testing refers to the interaction between the components of an application, more specifically to this case, the communication between the frontend and the backend, but also to the external APIs used. This was achieved through the verification of user-input data and its affect in the backend. The integration testing involves a top-down and bottom-up approach to track the data flow from one module to another. The top-down approach starts the testing from the highest-level modules, which as detailed in the system architecture chapter, is the user interface. The data submitted by the user through the POST requests is checked by the end points handlers, the validation algorithms and then on the database level. 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, in this case the database. Again, data integrity is checked on each level, to prevent faulty responses to the user. This approach was especially used concurrently with the development phase, as this also took the bottom-up approach.

## 6.3 Results Obtained

## 6.4 Future Plans

The first plan of extending the application is to include other urban vehicle regulations, which aim to reduce the pollution. Some of them are Zero-Emission Zones, road tolls or Emergency Air Pollution Schemes. 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 several new database tables for each regulation type. The access validation algorithms also need to be extended to cover these regulations.

For the moment, the application is exclusively designed for passenger car drivers. A future expansion can encompass all vehicle types, from motorcycles to heavy vehicles. From a database model perspective, such expansion involves the creation of some new database tables, to include each vehicle type and the addition of new fields in the *Zone* table, to incorporate the access requirement for each vehicle type. Also, small adjustments in the validation algorithms and the addition of a field where the user selects the vehicle type is mandatory. This update not only broadens the application's target audience but also creates the possibility of collaborating with enterprises for managing vehicle fleets. This can be easily achieved by continuing the research the same way it has been done so far and extending the database model and validation algorithms.

Future plans involve that the application encompasses a large number of daily users. This creates different possibilities for the extension of the system. One of them is the integration of a shop feature which would enable users to directly purchase registrations through the application. This additional function not only enhances user convenience but also diversifies revenue streams, thus increasing the application's sustainability. So far, the application has been designed to target regular individuals, but its features can be extended to accommodate the management of large enterprises. This would be a helpful tool for any companies, especially those activating in the delivery field, as it would allow the management of the vehicles and assigned registrations directly from the application.

Crucially, ongoing updates are imperative to ensure the application remains aligned with the ever-evolving landscape of LEZs across Europe and adapts to regulatory changes. Maintaining up-to-date information is imperative for the application's effectiveness.

# 7 Conclusions

Considering the current situation of the Low Emission Zones in Europe and the different restrictions imposed throughout the continent, navigation becomes more and more difficult. Vehicle regulations will persist and even multiply, especially in the urban settlements. Though they might be viewed as unnecessary or a punishment for some people, the studies indicate that these restrictions are not only effective in reducing the pollution, but also necessary. For sure, vehicle emission is not the only pollution factor, but in crowded cities affects people directly and can be easily diminished. I consider the existence of a universal platform for information to be an essential tool for navigation if modern times. This application must be accessible to everyone who wants to learn more about this topic and use the resource to make smart and informed decisions. In order to reduce the amount of time needed to research, which also improves overall morale, the application must use modern technologies to integrate the data and provide prompt answers to the user’s scenarios. This approach shifts the information structure from writing and editing long text paragraphs to the development of complex data processing algorithms. In other words, the user needs to know if his personal vehicle is compliant to the regulation or what actions are needed to ensure that.

The application presented in this document has successfully accomplished these goals and checked the requirements and specifications set during the analysis phase. The creation of this web app represents an online platform that facilitates users' access to information regarding Low Emission Zones across Europe. If more and more people understood both the regulations and the benefits that come with LEZs, the compliance would increase, thereby directly impacting the efficiency of these frameworks. Having a user-friendly interface, designed with intuitive features and impressive images contributes to a smooth and engaging experience for the end-user. The application manages to lower the barrier of entry in this field, which is essential in order to reach as many individuals as possible. On the other hand, the web application offers and a handful of technical advantages to the user. The possibility of account creation and vehicle registration, boost client loyalty and trust, thereby enhancing user retention and satisfaction. This lays a solid foundation and ensures a positive user experience, as individuals can quickly search for accurate information based on their specific circumstances and needs.

In conclusion, this application represents a first step towards creating a central hub for information regarding vehicle access and other urban regulations. The architecture of the application allows the implementation of future updates, like the addition of new regulation schemas. Additionally, for smooth system management, the dedicated Admin page allows technical users to make slight adjustments to the database model, in order to keep the information up to date with the ongoing LEZ changes. In conclusion, this application encompasses a fully functional access validation system, that brings real value and service to the European community.

# Bibliography

*[1] https://www.europarl.europa.eu/topics/en/article/20190313STO31218/co2-emissions-from-cars-facts-and-figures-infographics*. (n.d.).

*[2]* [*https://transport.ec.europa.eu/transport-themes/urban-transport/urban-vehicle-access-regulations\_en*](https://transport.ec.europa.eu/transport-themes/urban-transport/urban-vehicle-access-regulations_en)

*[3]* [*https://dieselnet.com/standards/se/zones.php*](https://dieselnet.com/standards/se/zones.php)

*[4]* [*https://www.mdpi.com/2071-1050/15/23/16260*](https://www.mdpi.com/2071-1050/15/23/16260)

*[5]* [*https://dieselnet.com/standards/eu/lez.php*](https://dieselnet.com/standards/eu/lez.php)

*[6]* [*https://www.germanemissionssticker.com/the-german-diesel-ban/*](https://www.germanemissionssticker.com/the-german-diesel-ban/)

*[7]* [*https://www.france.fr/en/holiday-prep/crit-air-anti-pollution-vehicle-sticker*](https://www.france.fr/en/holiday-prep/crit-air-anti-pollution-vehicle-sticker)

*[8]* [*https://urbanaccessregulations.eu/countries-mainmenu-147/france/paris*](https://urbanaccessregulations.eu/countries-mainmenu-147/france/paris)

*[9]* [*https://www.statista.com/statistics/1321264/low-emission-zones-europe-by-country/*](https://www.statista.com/statistics/1321264/low-emission-zones-europe-by-country/)

*[10]*[*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*](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)

*[11] Miguel Grinberg, Flask Web Development: Developing Web Applications with Python, O'Reilly Media, 2018*

*[12] Feng J, Hou H. Review of Research on Urban Social Space and Sustainable Development. Sustainability. 2023; 15(22):16130.* [*https://doi.org/10.3390/su152216130*](https://doi.org/10.3390/su152216130)

*[13]* [*https://www.w3schools.com/html/html5\_geolocation.asp*](https://www.w3schools.com/html/html5_geolocation.asp)

*[14]*[*https://www.statista.com/statistics/1241923/worldwide-software-developer-programming-language-communities/*](https://www.statista.com/statistics/1241923/worldwide-software-developer-programming-language-communities/)

*[15]* *https://survey.stackoverflow.co/2023/#overview*

*[16]* [*https://flask.palletsprojects.com/en/3.0.x/*](https://flask.palletsprojects.com/en/3.0.x/)

*[17]https://www.redswitches.com/blog/flask-vs-django/#:~:text=Netflix%3A%20Netflix%20is%20among%20the,Flask%20to%20build%20internal%20tools.*

*[18] https://www.sqlite.org/lang\_transaction.html*

*[19]* [*https://realpython.com/python-sqlite-sqlalchemy/*](https://realpython.com/python-sqlite-sqlalchemy/)

*[20]* [*https://discovery.hgdata.com/product/sqlite*](https://discovery.hgdata.com/product/sqlite)

*[20.5] https://w3techs.com/technologies/details/cp-javascript*

*[21]* [*https://react.dev/*](https://react.dev/)

*[22]* [*https://angular.io/*](https://angular.io/)

*[23]*[*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async\_function*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function)

*[24]* [*https://nodejs.org/en*](https://nodejs.org/en)

*[25]*[*https://aws.amazon.com/what-is/api/#:~:text=APIs%20are%20mechanisms%20that%20enable,weather%20updates%20on%20your%20phone*](https://aws.amazon.com/what-is/api/#:~:text=APIs%20are%20mechanisms%20that%20enable,weather%20updates%20on%20your%20phone)*.*

*[26]* [*https://www.altexsoft.com/blog/what-is-api-definition-types-specifications-documentation/*](https://www.altexsoft.com/blog/what-is-api-definition-types-specifications-documentation/)

*[27]*[*https://www.ibm.com/topics/rest-apis#:~:text=A%20REST%20API%20(also%20called,transfer%20(REST)%20architectural%20style*](https://www.ibm.com/topics/rest-apis#:~:text=A%20REST%20API%20(also%20called,transfer%20(REST)%20architectural%20style)*.*

*[28]* [*https://api.first.org/*](https://api.first.org/)

*[29]* [*https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods*](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods)

*[30]* [*https://flask.palletsprojects.com/en/2.3.x/tutorial/views/*](https://flask.palletsprojects.com/en/2.3.x/tutorial/views/)

[31] Permana, Iman, Mardi Hardjianto, and Kiki Ahmad Baihaqi. "Securing the website login system with the SHA256 generating method and time-based one-time password (TOTP)." Systematics 2.2 (2020): 65-71.

[32] <https://werkzeug.palletsprojects.com/en/2.3.x/utils/>

[33] Analiza si proiectarea sistemelor informatice

[34] <https://aws.amazon.com/what-is/ide/#:~:text=An%20integrated%20development%20environment%20(IDE,easy%2Dto%2Duse%20application>.

[35] <https://conda.io/projects/conda/en/latest/index.html>

[36] <https://www.redhat.com/en/topics/api/what-is-a-rest-api>

[37] <https://www.json.org/json-en.html>

[38] <https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=page-asynchronous-javascript-xml-ajax-overview>

[39] <https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model>

[40] Xia, Chuanlong, Guangcan Yu, and Meng Tang. "Efficient implement of orm (object/relational mapping) use in j2ee framework: Hibernate." *2009 International Conference on Computational Intelligence and Software Engineering*. IEEE, 2009.

*[41]* Chen, Tse-Hsun, 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.

[42] Gaspar, Daniel, and Jack Stouffer. Mastering Flask Web Development: Build Enterprise-grade, Scalable Python Web Applications. Packt Publishing Ltd, 2018.

[43] https://jinja.palletsprojects.com/en/3.1.x/intro/

*[44]* [*https://opencagedata.com/guides/how-to-compare-and-test-geocoding-services*](https://opencagedata.com/guides/how-to-compare-and-test-geocoding-services)

*[45]* *https://waqi.info/*

# Appendix 1. Code Snippets

@views.route('/new-car',methods=['POST','GET'])

@login\_required

def new\_car():

if request.method== 'POST':

owner\_id=current\_user.id

…

#registrations

france\_reg=request.form.get('France\_registration')

germany\_reg=request.form.get('Germany\_registration')

…

new\_car Car(owner\_id, … ,france\_reg,germany\_reg, …)

db.session.add(new\_car)

db.session.commit()

flash('Car added. ', category='Success')

…

return render\_template(“new\_car.html”,user=current\_user, …)

Ex. Code 1

@views.route('/info/<country>')

def country\_info(country):

country=country.upper()

# Here you can retrieve additional information about the selected country

# For example, you can query the database based on the country name

# Then render the corresponding HTML template for that country

from .models import Zone

list\_cities = Zone.query.filter\_by(country=country).with\_entities(Zone.city).distinct().all()

cities = [city[0].strip("()''") for city in list\_cities]

countries\_with\_lez =Zone.get\_countries()

cities\_by\_country=Zone.get\_countries\_and\_cities()

print(f"cities of {country}: ", cities)

return render\_template(f"/countries\_templates/info-{country.lower()}.html", user=current\_user,country=country, cities=cities, countries\_with\_lez=countries\_with\_lez, cities\_by\_country=cities\_by\_country)