### **A logo of a university Description automatically generated with medium confidenceA blue and yellow badge with a picture of a building and a gold ribbon Description automatically generatedUniversity of Bucharest**

**FACULTY OF MATHEMATICS AND INFORMATICS**

**DEPARTMENT OF INFORMATICS**

**Bachelor's Thesis**

**HistoryMapper**

**Graduate**

**Neaga-Budoiu Maria**

**Scientific coordinator**

**Lect. Dr. Dobrovăț Anca Mădălina**

**Bucharest, June 2024**

#### **Abstract**

History has always been a major subject for creating a solid basis of general culture and exploring it could be facilitated using technology. This is the main objective of the HistoryMapper application, to make the process of learning History more accessible and appealing using technology, and, in particular, to help the exploration of historical events by forming visual connections between their position in time and space.

Therefore, the designed platform provides an interactive map, obtained with the usage of Google Maps API. It changes through a timeline axis and highlights historical events and routes manipulated using different local APIs. Even if the map is the core of the application, other important features are provided as well, including the possibility for users to navigate through historical periods, verify additional information about all the facts and the eras, or to participate in quizzes related to the events they discover on map.

The result of this project is the web application whose back-end is written in Django framework for Python, while the interface is created in Angular. The data used by HistoryMapper is stored in an Oracle SQL database, which is partially populated with information fetched from the storage system for Wikipedia platform.

#### **Rezumat**

Istoria a fost întotdeauna un subiect principal în crearea unei baze solide de cultură generală și explorarea ei poate fi facilitată de folosirea tehnologiei. Acesta este principalul obiectiv al aplicației HistoryMapper, să facă învățarea istoriei mai accesibilă și atractivă cu ajutorul technologiei și, în particular, să ajute la explorarea evenimentelor istorice prin crearea de conexiuni vizuale între localizarea lor în timp și spațiu.

Astfel, platforma dezvoltată oferă o hartă interactivă, obținută prin folosirea API-ului de Google Maps, care se modifică printr-o axă temporală și evidențiează evenimentele și rutele istorice manipulate cu ajutorul diferitelor apeluri de API-uri. Deși harta este nucleul aplicației, alte funcționalități sunt, de asemenea, oferite, inclusiv posibilitatea ca utilizatorii să navigheze prin perioadele istorice, sa verifice informații adiționale despre toate faptele și erele sau să participe la quiz-uri legate de evenimentele pe care le descoperă.

Rezultatul acestui proiect este aplicația web al cărei back-end e scris în framework-ul de Python Django, în timp ce interfața e creată în Angular. Datele folosite de HistoryMapper sunt stocate într-o bază de date Oracle SQL care e parțial populată și cu informații preluate din sistemul de stocare al platformei Wikipedia.

# **Contents**

**1** [**Introduction** 5](#_Toc167718267)

[1.1 Motivation 5](#_Toc167718268)

[1.2 Concurrency 6](#_Toc167718269)

[1.3 Exploited Domains 9](#_Toc167718270)

[1.4 Project Structure 9](#_Toc167718271)

[1.4.1 Data Manipulation 9](#_Toc167718272)

[1.4.2 Map Feature 10](#_Toc167718273)

[1.4.3 Users’ System and Actions 10](#_Toc167718274)

2 [Data Manipulation 11](#_Toc167718275)

[2.1 Database Structure 11](#_Toc167718276)

[2.2 Back-end APIs 12](#_Toc167718277)

[2.3 Google APIs 14](#_Toc167718278)

[2.4 Historical Periods 15](#_Toc167718279)

[2.5 Populating the Database 17](#_Toc167718280)

[2.5.1 Manually 17](#_Toc167718281)

[2.5.2 From DBPedia Database 19](#_Toc167718282)

[2.5.3 Automatic Location Mapping 20](#_Toc167718283)

[2.5.4 Automatic YouTube Videos Insertion 21](#_Toc167718284)

3 [Map Feature 23](#_Toc167718285)

[3.1 Map Integration 23](#_Toc167718286)

[3.2 Timeline Axis 24](#_Toc167718287)

[3.3 Events on Map 25](#_Toc167718288)

[3.3.3 Markers 25](#_Toc167718289)

[3.3.4 Search and Filter Options 28](#_Toc167718290)

[3.3.5 Historical Routes 29](#_Toc167718291)

[3.3.6 Clustering Algorithm 30](#_Toc167718292)

4 [Users’ System and Actions 32](#_Toc167718293)

[4.1 Authentication System 32](#_Toc167718294)

[4.2 Admin Actions 35](#_Toc167718295)

[4.3 Quizzes 37](#_Toc167718296)

5 [Technologies Used 40](#_Toc167718297)

[5.1 Back-end 40](#_Toc167718298)

[5.2 Front-end 42](#_Toc167718299)

[5.3 Database 43](#_Toc167718300)

6 [Conclusions 44](#_Toc167718301)

[6.1 Summary 44](#_Toc167718302)

[6.2 Further Development Possibilities 45](#_Toc167718303)

[Bibliography 46](#_Toc167718304)

# **Introduction**

HistoryMapper is defined as a web application for exploring History, focused on providing a temporal space context for a wide range of events of the past. It binds together dynamic maps with historical information, accentuating its educational functionality through events-based quizzes, and presenting its features through an interactive interface.

What I consider that makes the project a complete web application are its three main architectural components, which communicate with one another: the back-end side, the front-end side, and the relational database. Since I designed HistoryMapper to be based on API methods, they are the ones that link together the back-end with the interface, and they make the platform aligned to the current technologies by calling some popular external APIs.

In this document, I will go through the purposes of my application, analyzing the process of making History learning a domain of constant actuality. I will also describe the steps I followed in its implementation and highlight the variety of features it provides, which leave space for future development possibilities.

## **Motivation**

History has always been a subject of interest for me, and I consider that today’s younger generations are losing interest in it because of the lack of modernization of the methods for its learning. Moreover, the multitude of information that can be generally found in historical sources along with the difficulty of structuring facts could become confusing. A solution for these aspects I believe can be brought with the use of technology, making a domain sometimes considered outdated more attractive for those who explore it. Having this in mind, I have chosen an interactive and facile approach for presenting historical facts in my web application.

HistoryMapper is addressed to anyone who wants to explore the domain, from historians or teachers who need quick access to data to users who intend to discover facts about the past. Therefore, the platform also has an educational utility as it provides information materials for the studying process, including data about the events of Europe and all the periods of Human History, and quizzes that test the knowledge accumulated by users.

I decided that the platform should be centered on the map feature because it creates visual connections between the historical timeline and the physical locations of the events, which serves the objective of an interactive application and the creation of a suggestive navigation system. Apart from the fact that visual learning is a powerful method for memorizing information, I also believe that this way of presenting History is not much explored by other sources and has a lot of potential to be further exploited.

## **Concurrency**

Like many other existing applications, HistoryMapper competes with the AI systems, such as ChatGPT, which are in a continuous evolution. But what my application brings different in the process of discovering historical data I consider is the interactive interface, the mapping of events and routes on a physical map, and, nevertheless, the quizzes system. While AI models are able to obtain or filter any information present on the Internet based on the requirements of the user, they do not include methods of displaying it and visually connecting it to other related data in order to form associations between various facts. Moreover, they do not include an interactive system of analyzing events: for example, in HistoryMapper, the temporal context of the events selected can be quickly changed by user interactions with the timeline axis, and the visual context changes dynamically, while selecting events in a different time range in ChatGPT implies a new written request. To this aspect is added the fact that, in my application, additional information for a specific event is accessible by simply clicking on its marker, leading to a much faster and accessible navigation system through History. Other strengths provided by the current project are logical classifications of events by type and category, clearly established relationships between data entities obtained via the structure of the database, and an appealing, user-friendly design that makes the overall experience more attractive.

Among the projects which are similar to HistoryMapper, there can be named OpenHistoricalMap, an open-source platform for visualizing political maps of the continent which change throughout time that also contains a timeline axis. What makes HistoryMapper different from it is that it displays events, and not changes in political organizations of the continents. To this fact is added the one that the timeline in my project brings dynamic changes at any user interaction, while the axis of the other application requires selection of a time range through buttons. Furthermore, OpenHistoricalMap does not include additional information regarding the periods of History, nor a system for quizzes.

A map of europe with a blue background

Description automatically generated

*Figure 1.1 Main page of OpenHistoricalMap [1]*

Another similar platform is HistoryPin [2] that allows users to add historical references to a section of the map, including related videos, images or descriptions. However, my application includes some different functionalities: the map can be checked as a whole and not only by a specific area, the events cover all the time periods, the accent is not put on the local traditions and discoveries of a selected place, but rather on events at a larger scale, and there exists the option of visualizing routes. I consider that HistoryPin is more focused on letting people share parts of the local History of their places and communicate with other users, while HistoryMapper enhances the educational functionality and the cursivity in event succession.

A screenshot of a map and a building

Description automatically generated

*Figure 1.2 Navigation page of HistoryPin [2]*

TimeMaps is also serving the purpose of navigation through History and is an open-source platform with integrated maps. Even though it provides information for eras, and materials for teachers and students which makes it similar to my project, it does not include dynamic maps as the selection of time does not allow an interval, but only one year, and each selection opens a static map. Users cannot navigate the maps since they are images with marker icons. What is more, the accent is put on the civilizations of the world, and there is no information added for singular events. The historical facts are presented on a larger scale, being characterized by the category they belong to as a whole. Other different functionalities HistoryMapper has are the routes system, the quizzes, and the integration of videos.

A screenshot of a website

Description automatically generated

*Figure 1.3 Main page of TimeStamps [3]*

## **Exploited Domains**

For creating this project, I considered approaching domains covered in my academic experience. I chose to focus on:

* **Web Applications Development,** used for creating the overall project structure, the API endpoints, the authentication system and user actions, the front-end elements, and the connection with the back-end, and the database
* **Software Engineering,** highlighted in establishing the goals of the application, as well as the development process, organized on tasks with different priorities scheduled over a period of time
* **Databases,** for the creation of the models and the relationships between them, and for the queries used in the back-end APIs
* **Object Oriented Programming**, applied to the creation of the API classes, as well as to the logic developed in the Angular components and services

## **Project Structure**

The project is roughly composed of the following parts: the aspects regarding database manipulation and completion, the elements related to the map feature, and the system of users with the corresponding actions. In the sections below, I will describe the development aspects regarding each of these sections, including the addressed approaches and the technologies used, and I will also accentuate the results obtained at each step.

## **Data Manipulation**

One of the most important particularities that I established for the application from the beginning was the possibility to store a large number of European events in order to cover all periods of Human History as much as possible. This is the reason why the Oracle SQL Datbase used as a storage system has as a center-entity the one with events data, which is connected to the entities storing locations and historical periods, both essential for placing events in a tempo-spatial context. In this section I will cover one of the major parts of the development process, data creation and handling, which will include the logic of the application’s APIs, the connection to DBPedia database, the storage of historical periods, and the automatic population of the locations entity with the algorithms for establishing geocoordinates.

## **Map Feature**

The creation of this functionality involves all the operations for personalizing and manipulating the map, from correlating it to the timeline axis to features such as searching and filtering events on map or displaying historical routes. For this section, I will also explain the process of Google Maps API integration for displaying the map, and the usage of different Google Maps classes for markers and routes. One other aspect that will be covered is the clustering algorithm, a method for grouping events based on zoom level of the map. Nevertheless, the platform has integrated YouTube videos that complete the information about the events on the map, so I will describe the method for calling the YouTube API and obtaining them.

## **Users’ System and Actions**

The platform is intended to be user-oriented, so I created a system for authentication, which also provides access for users to the quizzes functionality. The performance data for each quiz will be stored per-user, logic which was created with the help of a quiz history entity in the database. Apart from the access to questions, the authentication functionality also gives the administrator of the database access to special CRUD actions for events, including the automatic population of the Event entity.

# **Data Manipulation**

## **Database Structure**

All the information for the application is kept in a relational database of type Oracle SQL, linked through a connection to the back-end part of the project. The connection is established in the *settings.py* file of the back-end application by specifying the required database parameters.

Django is a powerful framework for Python which provides a facile mechanism for handling data through the models, sources of data simulating the tables in a database.[4] When a model is created, a new entity matching it is added automatically to the database along with the corresponding constraints. The conventions are the following: for the tables which are related to the authentication system, the tables in Oracle SQL with have names starting with the *auth\_* prefix, while the others will begin with the name of the back-end application, in this case *explore*. Although these are the real names of the database tables, in this document I will refer to them with the name of their corresponding models for simplicity reasons.

Another useful particularity that Django provides are methods for executing SQL Queries in the Python code through an ORM, facilities that made the code of the application more compact and the development process faster.

HistoryMapper’s database is focused on the Event entity, that contains all the data needed for the historical facts to be placed on the map. This table is directly connected to almost all the other ones, for having access to additional information about the events such as the location, the historical period or the category they are part of.

The ERD (Entity-Relationship Diagram) schema of the database[[1]](#footnote-1) is presented in Figure 2.1, as it was constructed in the DrawSQL platform [5].

A screenshot of a computer

Description automatically generated

*Figure 2.1 ERD Diagram of HistoryMapper*

The Event entity contains foreign keys (references) to the IDs of the location and period they are part of, the type (battle, treaty, truce and so on), and the category. While the type is introduced to make identifying an event easier or to group facts in a visual manner in the platform, and is used for the filter option, the category is required for future automatic insertions and for the creation of quizzes, functionalities described in the following chapters.

Apart from the entities for historical facts, the database also includes the users’ system. Because Django already provides such a system through a specific library (*django.contrib.auth*) imported in the back-end, the User model inherits the already built-in class with the same name, and is necessary only for the creation of a mapping table in the database. This entity is linked to a table storing a quiz result history related to each registered user.

## **Back-end APIs**

All the pages in the platform handle information from the database under different forms. This logic; either being fetching data, modifying the entries or creation of new ones or delete operations; is done in the back-end part of the application in methods called APIs. They define a specific logic for handling the data required by the front-end, and each time the resource is requesting an operation, one API is called and retrieves a needed response or a punctual error message. For example, displaying the events on the map requires the existing database events in order to place them as markers, so the page makes a request to obtain this resource. Not only do these back-end methods obtain data, but they also transform it to a desired format and add additional elements to it if they are needed in Angular. The code includes specific status messages in all the responses, each one with a particular meaning. In general, for operations completed successfully, the status will be 200, for resources that are not found will be 404, and for internal server errors 500, but other codes can occur depending on the specific situations. This whole process of adding and intercepting the status of the actions is obtained with the usage of the status library from *rest\_framework*.

Actually, in the implementation of HistoryMapper, I created all the APIs using REST (Representational State Transfer) framework, an architectural style defining constraints to be used for creating web services [6]. For this, I installed the Django REST framework package, which also brings as a benefit a user-friendly interface for testing each API. The logic imposed by REST framework implies that all the methods are of one of the types GET, POST, PUT or DELETE, depending on the type of request the client (the Angular application) sends to the server (the Django part). All API calls are HTTP requests sent to the server via a unique web URL, and the response is then sent back to the client in JSON format. I chose this format not only because it is the most popular for web services today, and also because it is easy to access the fields of the response back in the Angular application. For more information about the usage of REST framework, consult the chapter regarding the technologies used in HistoryMapper.

One important API for the platform is the one retrieving events between two given years because it fetches the data for the map page, later used by many other functionalities such as clustering, filtering, searching, or establishing routes for events. It is called every time a change in the timeline axis occurs. Four parameters are sent to the server via the URL, the start and end years and eras (AD or BC). Diving deeper, into the implementation of the API, it is created as a class containing a method of type GET, because it should only retrieve data from the server.

The method converts years BC to negative values, action necessary for executing the SQL query that will retrieve entries from the Event entity, performed directly in the Python code thanks to the object raw method supplied by Django. After that, the method calls a global function for getting coordinates for each event and populating the MapLocation entity, converts all the data to JSON format and returns it as the response to the client.

Similar to this implementation are all the other GET functions in the API classes. However, the methods of types PUT and POST will receive data to be updated, respectively added to the database via a parameter called request explicitly passed to them when the APIs are called in Angular. This parameter simulates an entire entry information for the database and thus cannot be specified in the URL as the parameters like the ones described in the paragraphs above. The last type of method, DELETE, is used in HistoryMapper only for removing an event or a user from the database and requires its identifying information, making the method similar to the GET ones.

Endpoints of different types can be placed under the same API (and thus under the same URL as well), so how does the application know what method to choose? The answer is that the selection depends on the format and type of request sent from the client.

Now, turning to the client side, I will briefly describe the structure of the Angular project, with accent on the API calls. A more detailed description is included in the chapter destined to the technologies used in HistoryMapper.

The front-end side consists of components, each handling a different page or section of the platform. They have access to the APIs in the back-end and to some parameters shared between components via services, which are classes with private fields, public getters, setters and other methods. The access to the services from components is realized via the principle of Dependency Injection, so the service with the needed API call is injected into the constructor of the component. After that, the service method that fetches API responses is called in an asynchronous manner with the async-await mechanism, and the result of the API call is saved in component for use.

## **Google APIs**

HistoryMapper utilizes various Google API endpoints in its implementation, either in the local APIs found in Django [7], which are presented in the section above, or directly in the front-end, for the map page. They serve different needs of the platform: displaying a map, obtaining geocoordinates, fetching YouTube videos and so on.

Users have access to Google APIs only after logging into the Google Cloud Platform for Developers and requesting a private Google API Key. This one must be linked to the platform projects where APIs will be enabled [8]. I firstly created a new project in the Developers Console [9], and added the private key used by it in HistoryMapper’s database because it is required by the method calls inside the code of my project, yet its value must be kept private for security reasons. However, the key can be safely fetched from the application’s storage system.

Google Maps JavaScript API is the one that permits the display of the project’s map, as well as the creation of the individual map markers or of the clusters, and the presence of historical routes as roads connecting map points. Information about the structure of the API’s requests and responses, as well as the methods for its integration into applications, is found at its official documentation page [10].

Another such method is the Google Geocoding API [11], responsible for converting locations into physical coordinates (latitude and longitude) and vice versa, and it is useful for determining the map geolocation of the historical events in the project. So, the location of the event will be sent by the client (in this case, the Django application which calls the API) as the value of the address field in the URL, and the response will include the coordinates. One requirement here is that the address should be specified in a particular format: the name of the locality followed by the prefix of the country, the reason why this structure is found for the names of the locations in HistoryMapper’s database as well.

Apart from these Google methods, the YouTube API, version 3, is also included in the project, for fetching links and thumbnails for videos related to the events. I used the functionality of searching for content by specific topics, which is documented on the official YouTube API page [12]. The result is composed of the video data returned in JSON format, to be processed in the Angular application and displayed for each event on a dedicated page.

## **Historical Periods**

For highlighting the educational particularity of HistoryMapper, I wanted to give users access to information about ten periods of time covering almost all the Human History from the prehistoric times to the present [13]. All the events fall into one of these periods based on the date they took place in, so I consider that additional information resources about the historical eras could offer more insight about the explored events as well and give users the impression of a journey through time.

The HistoricalPeriods front-end page displays these eras in chronological order after getting them via an API call. A section of this page is presented by Figure 2.2. In case of a click on one of the ten sections, a new page is opened, showing information regarding the selected period. But how does the page with historical period information know what period it should display data about, and, more specifically, what period was selected in the HistoricalPeriods page? The answer is that the two components, the HistoricalPeriod and HistoricalPeriodInfo, share via a service a common parameter, the ID of the currently selected event. So, when the user clicks on the section for a specific era on the first page, a private ID field in the periods service is set to the unique key of that selected historical period. When the new page, the one with detailed description, loads, it gets the saved ID value from the same service, fetches a period by ID with the call of an API, and displays the data in HTML. This is how a field is shared between different components, the value is kept in a private field in a service injected in the components, and the data is handled with public getters and setters.

A screenshot of a computer

Description automatically generated

*Figure 2.2 Section of the HistoricalPeriods page*

One last feature related to this part of HistoryMapper is that each historical era can be visually seen on the map. The user can navigate directly from HistoricalPeriodInfo page to the map and see the events of the selected period displayed. To create this, I made the values of the timeline axis change to the period’s interval of time whenever the user clicks to be redirected to the map from the page of the section dedicated to a specific period. The redirection is executed via the routing system of Angular, and sharing the values of the interval between the components is obtained with the service for handling the behavior of events.

A screenshot of a website

Description automatically generated

*Figure 2.3 Section of the HistoricalPeriodInfo page*

## **Populating the Database**

For storing a large quantity of events in the database, manually populating it entirely was not a good option, and I needed to find a method to add more entries at same time automatically. This represented a real challenge especially because the Event table contains a lot of foreign references to other tables (Category, EventType, MapLocation, HistoricalPeriod). This fact leads to the necessity of automatically adding correspondent entries in some of the linked tables as well, for example inserting new locations for the new events to be stored in the database. But the fact that additional information about the events must be stored is not the only reason other entities must be populated with a dynamic algorithm: some information, such as geocoordinates for the map or YouTube videos for events, have to be obtained with the help of some Google APIs, so the process of calling them can be regarded as an automatic data creation. In this section, I will go through the process for populating the database, from the data which was added manually to the one inserted via back-end algorithms.

## **Manually**

Populating the database manually was the first chosen method because, along with other considerations described below, I also needed some initial data to begin the first stages of implementation. However, this approach led to an inefficient way of working, speed compromises, and also involved some shortcomings for some tables in the database.

One entity which was populated entirely manually is the HistoricalPeriod because I decided to split the human History into ten main eras and thus the amount of information is small. Another table with the same characteristics is the EventType, including only sixteen main types the historical facts can fall into. In the first stages of development, I needed to have some test data for other tables as well, so I manually added some entries in Event, MapLocation, and Category as well, based on historical references from various sources [14]. Some of the tables do not appear in this enumeration because of several reasons: the User entity depends on the authentication system, so it is basically populated at log in, the QuizHistory depends on the users’ actions on the quizzes page and is populated automatically, and the Video table is strictly related to data fetched by the YouTube API.

Adding new entries by hand can be done either with SQL queries written in a database console application or through the interface provided by Django. I opted for the second choice because it is a faster process as the Django template project includes a page where the admin of the application can add entries on the interface. The only required aspect is that the models be included in the *admin.py* file of the project’s structure in order to appear on the interface. After that, the Django application is running on port 8000 and the required page is accessed at */admin* by entering a previously chosen username and password.

The admin interface Django offers is shown in Figure 2.4.

A screenshot of a computer

Description automatically generated

*Figure 2.4 The admin interface in the Django Application*

## **From DBPedia Database**

DBPedia is a project for handling structured information from the Wikipedia platform and it makes it available on the Internet. Users can query properties of chosen Wikipedia sources and obtain them in a given format, which is described on the DBPedia official webpage [15].

In HistoryMapper, obtaining historical events from DBPedia to be inserted in the local database is done by specifying the Wikipedia category they belong to, which is different from the actual category, the one they will be mapped to in the SQL database. I decided not to insert categories with the same name as the ones in Wikipedia because they can be too restrictive in some cases and the application should have event categories that characterize a wider range of database entries.

Now, one challenge in the process of adding events from DBPedia is transforming the information to the format required by the local database. To this aspect is added the fact that events of different types are retrieved in different formats from DBPedia, and these issues led to the necessity of creating separate functions for types of events that differ. I chose to write three individual functions that fetch the data for battles, treaties, and successions from Wikipedia, but the application can be further developed to obtain entries for the other types as well, by adding new similar functions.

In my project, I opted for the Public SPARQL Endpoint for DBPedia which can be accessed via the [*http://dbpedia.org/sparql*](http://dbpedia.org/sparql)URL. The query language used is thus SPARQL, effective in retrieving data across various web sources in a specific format called RDF. This format is organized as a directed, labeled graph structure, so one important step in the process is transforming the RDF data into a JSON response. A complete description of the SPARQL language can be found at the page of its documentation [16].

As a first operation, I imported the SPARQLWrapper library in my project to be able to execute SPARQL queries directly in Python. In each of the functions calling the DBPedia API, the endpoint is wrapped, and then there is created the query for retrieving data by Wikipedia category, query that is, after this, executed and its results converted to JSON. But, for example, the information retrieved for battles included in a certain category is not the same as the one obtained for treaties, even if they belong to the same historical group in Wikipedia. What is more, some results of the query do not define complete future entries in the local database and some fields might be missing. So, the information must be completed and turned to the desired format before retrieving it, which is an operation handled differently by each of the three functions.

For all the event types, one example that illustrates that some data might be missing is the one of the event locations found in the response of the SPARQL query. Sometimes the position of an event is specified as a string name when the actual needed data are the geolocation coordinates. In such case, another query is created and run for converting the Wikipedia location to actual latitude and longitude values.

The requests for populating the database with data from DBPedia are sent from the front-end of the application, so, in the back-end, there must be established what type of events are added, what is the Wikipedia category they are fetched from, and what is the new category, the one in the local database, they will fall into. These values are passed as parameters of the request in the URL of the API endpoint that adds data, and based on the specified type (either battle, treaty, or succession), the corresponding function that fetches DBPedia data is called. Its returned response is processed in the API method: the date is turned into a format accepted by Oracle SQL, and the foreign keys to Category and EventType are obtained. In case the current value of the category does not point to any entrance in the database, a new category is inserted. More than this, the location names are transformed to the structure that is needed by the Google Maps API: the name of the locality followed by the prefix of the country.

Only after all these operations, the API method inserts new rows in the Event table with the obtained Wikipedia returns them as a response message back to the client.

## **Automatic Location Mapping**

Each event in the database needs to be placed on the map at the historical location where it took place, represented by a latitude and a longitude value. The problem is that multiple events could have taken place at the same location, meaning that, to avoid repetition in the database and, thus violation of correct database creation principles, the geocoordinates are not stored in the Event entity. Instead, a many-to-one relationship is established between two separate tables, Event and MapLocation.

When adding events into the database in an automatic manner, and even when they are inserted manually, mapping them to locations by hand can be difficult either because of the large quantity of events or to the fact that many-to-one relationships must firstly be checked. To this aspect is added the fact that the process of individually and manually obtaining the geolocation for all the places of the events is a slow process which can be much improved with Google API calls. Therefore, HistoryMapper’s back-end side contains methods that fetch geocoordinates for locations, and then add them to the storage system in case it does not yet include them.

Each time a new event is to be used in the response of an API, the application verifies if its location is already stored in the MapLocation table and, otherwise, adds a new entry consisting of the name, latitude, and longitude of the respective place. I decided to perform this check every time events are to be returned as a response and not at the moment of event creation because this provides additional assurance that no event location is missing before performing operations with them, such as trying to display the events on the map.

The logic of populating the MapLocation entity is divided into three main parts: establishing the new locations to be added, obtaining their geocoordinates, and adding them to the database.

The first part is implemented in a global function that will be called by all the APIs that retrieve certain events, and it receives as parameter a list of events that should have correspondent locations in the database. The implementation can be checked in Figure 2.4. I created a set of the names of all these locations, another one of the names already stored in the database, and then made a difference operation between them, with the purpose of obtaining the missing data. The coordinates need to be calculated for all these locations, so, in a repetitive manner, the logic for providing geocoordinates is performed.

For the second part, I created another global function where I used the help to the Google Geocoding API, as it is shown in Figure 2.5. The API call is done via an URL including the private Google API Key value and the address of the event to be converted.

Lastly, back in the initial function that populates the database, after the process of obtaining coordinates is called for each new location, the entries to be inserted are composed with the fields of name, latitude, and longitude, and saved one by one in the database.

## **Automatic YouTube Videos Insertion**

The final automatic algorithm for database population is the one involving YouTube videos for the events. Given the fact that there are many events in the application, and manually mapping videos for each of them is difficult and could cause data errors or mismatches, both the entries in the Video entity and the relationships to the Event rows are created dynamically in the back-end.

The process is the following: the POST API endpoint receives as a request the name of the event for which videos should be fetched. If this event already has videos associated to it, then these entries are obtained with an SQL query and returned as a response. Otherwise, the YouTube API should be called, but not before constructing the resource to communicate with the YouTube server. This resource is constructed with the build function included in the *googleapiclient.discovery* import, and the connection established using the private Google API key. This is followed by the construction of the request with some specific parameters, including the topic (the name of the event), the maximum number of videos per event (five), and the type of results needed (videos). After executing the query, the results in JSON format contribute to the creation of the database rows, involving the construction of the final links with the IDs of the videos, and the selection of images.

# **Map Feature**

## **Map Integration**

Navigating along time and discovering events is a functionality elaborated in the application with the help of an interactive map which changes dynamically. So, the first step in the process of developing this feature is obtaining a map to be displayed on the dedicated front-end page, that will later be added the desired design and properties. In HistoryMapper, I chose to integrate a Google map because of more considerations. First, Google Maps is one of the most popular systems of navigation, fact that could make the map familiar to many of the users. Secondly, the user interaction with the icons of the map is suggestive and simple, which contributes to the user-friendly property of the platform. And, not least, it provides a wide range of marker and routes options, and design properties which can be customized for any element of the map (the water, the land, the localities, and so on).

The process of map integration into the Angular part of the project involved calling the Google Maps JavaScript API. It helps create an instance of the class Map, whose constructor generates a new map inside a given HTML container. However, I discovered that this class alone does not allow a process of setting values for the properties of the map (the zoom level, the center, and the specific id), so I had to make it implement the MapOptions interface containing all of them.

The zoom level and the center values were established from the beginning, but I introduced the id property only after developing the routes functionality. This happened because I wanted to create a differently personalized map when the routes mode was selected by the user. While separating the design elements from the declaration of the map worked fine when I had only one design of the map, introducing the routes mode design caused issues because the display properties could not be changed. I decided to approach the problem by creating two maps with the desired designs in the Google Cloud Console project, and pass their

id as a parameter to the function creating the map object. So each time the user switches from exploration mode to routes mode and vice versa, a new map object is now created, having the design of the map from the console with the same id as the parameter. I consider that this method eases the process of updating the design (that can be quickly done in the console interface), and also permits the introduction of new map designs in case of new updates of the platform’s interface.

Apart from the parameters, I also used methods included in the Map class for actions such as adding listeners to the map and later to the markers, for customizing the markers and placing the event points at their specific coordinates. The listener for the map is an operation triggered at zoom level and is correlated to the processes of creating markers for the clustering option, so I will explain the process in detail in the section destined to the clustering algorithm.

Other classes from the Google Maps JavaScript API that I introduced into the project are AdvancedMarkerElement [17], PinElement, and Polyline [18], used for either the markers or the routes on the map, which will be described in detail in their destined sections.

## **Timeline Axis**

The navigation through time periods is one of the most important particularities of HistoryMapper. I wanted to make the selection of historical periods and the changing of the map as straightforward as possible, so I opted for an axis with two ends that allows selection of time intervals and instantly generates the change of the map without additional operations. It is shown in Figure 3.1.

Angular framework confers a simple way of displaying an axis-like element through the range slider (the ngx-slider directive), basically a slider with two ends that controls the minimum and maximum value of the time interval [19]. The axis of the platform covers all the years from 3800 BC to 2024 AD (the values of the two ends) with the step between them equal to 1. All these values are mentioned in an Option structure which is passed to the Angular slider at its creation in the HTML page. In order to represent both BC and AD years on the axis, I considered the years of the previous era as negative values and created the association with their real values through a translate function. So, for example, the value of -1 is equivalent to the year 1 BC, which is displayed on the page, even though the negative value is the real quantity used. When the page responsible for the map feature loads, the axis takes some initial default values, established to mark the interval between 100 BC and 100 AD. I opted for these values as they mark a relatively small interval to start with which is, at the same time, visible enough to suggest to users the way of handling the timeline axis.

A black screen with white text

Description automatically generated

*Figure 3.1 Timeline axis with initial values*

Each time a user change is performed in the slider, the API for retrieving events between two years is called, and the response is prepared to be displayed on the map. The steps for calling the API are as follows: in case of a user change, the function for submitting years is called, which, in its turn, awaits the response from the function in the service that calls the Django EventsBetweenYearsAPI. As all the other API calls performed in the front-end, they are done with the async-await Angular system, meaning that the response is awaited by the client before moving to the next code instructions. The response is a list of events, further processed in the submission function and passed to be transformed into markers for the map.

## **Events on Map**

## **Markers**

Thanks to the back-end API which gets all the European events in a time interval, all the necessary data is now obtained in the Angular project, and the next step is creating the correspondent markers. Based on the zoom property of the Google map, different markers will be designed, so the logic is separated into two different methods. One of them is destined to the creation of markers for individual events, which are placed on the map only if the zoom level is a value greater than 4. Otherwise, the events will be clustered in groups, represented by another type of markers, therefore handled by a different TypeScript function.

The individual markers are instances of the AdvancedMarkerElement class, made available by the Google Maps JavaScript API. I initially intended to generate the markers as instances of the Marker class, but Google announced it will be deprecated starting February 2024, and should be replaced by the new alternative, the AdvancedMarkerElement.

I imported the *marker* library, then, for each event to be displayed, I specified some needed elements: the coordinates of the event’s location, an info-window with the name and date (that will be opened on hover over the marker), and an image to be overlapped on the marker. The last one is chosen based on the type of event the marker belongs to, each one having a corresponding image, bringing an attractive design to the map page, and helping the user differentiate the historical facts. Having these elements at hand, there is instantiated an AdvancedMarkerElement object that will have listeners (methods triggered by an event) added to it: in case the mouse is placed over the marker, the small info window will be opened, and when the mouse leaves, it will be closed. After adding these two listeners, each marker is finally pushed on the map. The result is presented in Figure 3.2.

A map with red pins with different locations

Description automatically generated

*Figure 3.2 Markers for individual events*

Separately, the markers for clusters are instances of the PinElement class, also an Advanced Element introduced by Google JavaScript API. The data fetched for this type of icon is done via another API, the one gathering information for groups of events, which is described in the section regarding the clustering algorithm. One difference is that the coordinates of these markers will be the ones of the centroids of each cluster. Even if the creation of these types of markers is similar to the previous one, there is the exception that, this time, the icons have a specific label, useful for displaying the number of events in each of the clusters, and the listeners mentioned in the last paragraph are no longer added. The zoom level when the page first loads is smaller than the value 4, meaning the initial markers will be of cluster type, but they are replaced by the other choice if the zoom is increased.



*Figure 3.3 Markers for clusters*

Whenever the user clicks on an individual marker and is not in the routes mode, an information section containing an event description and YouTube videos associated to it is opened. The YouTube videos are fetched with the method that can either populate the database with events with the Google YouTube API or just return the already existing ones. A separate component is responsible for this section, and its opening depends on an on-click listener attributed to the markers.

Some listeners, such as the ones opening the special info section mentioned above, depend on the saved mode of the page, and will not be added if another mode is selected. For example, the info section should not open when the user switches to routes mode, but it should be visible for the exploration (default), search or filter modes. However, the routes mode comes with other listeners for the marker instances. This is why this whole operation is handled in a function strictly serving the attribution of special listeners to the markers, dividing the logic and treating each different situation.

There are cases when, because of the large quantity of events, the map becomes too crowded, or the performance is affected. Considering this, I decided to reduce the length of the list to a maximum value equal to 500 and send a warning message in the interface suggesting narrowing the interval of the timeline axis.

One other aspect of the markers I considered in my project is their behavior to collisions. More events could have the same coordinates, but all the icons must be visible on the map, thus they must not overlap. For fixing this behavior, at the time of preparing the events to be displayed, I created a dictionary structure to store the number of collisions for each location, so the names of the places are the keys, while the number of events that collide are the values. For each event to be displayed, there is performed a check to see if its current location is a key in the dictionary, and, if so, then the value is incremented by one, otherwise a new key is inserted with the corresponding value equal to one. So, encountering collisions means having values greater than the value one in the dictionary. In such case, the markers for events that collide are slightly shifted from their original location by increasing the latitude and longitude with a small value. It is not an arbitrary value, but one based on the current value found in the dictionary which, more exactly, represents the index of the current collision in its location’s list.

## **Search and Filter Options**

Users can visualize some events they are looking for in a faster way than just by selecting the time period from the axis: they can use the options provided by the left menu on the map page. One such option is to search for an event by name which will appear alone on the map. Another approach is to filter the events on the map by their type: in case the user is looking for one or more battles, for example, they will select the specific filter from the list in the menu and only battle-type events will be kept on the map. Both methods represent a change in the mode of the map, so the Angular component will know what user request to serve by checking the value of the selected mode.

A screenshot of a phone

Description automatically generated

*Figure 3.4 Menu for map options*

But what exactly does the mode system mean and how does the selection occur? The process is handled in the Angular component responsible for the map menu. Depending on what option is selected by the user (search, filter or routes visualization), different operations are performed: if the choice is either search or filter of map events, then the menu expands to allow the user to specify further information.

When the user wants to search, a specific text bar appears in the menu, where the keyword needs to be introduced. In essence, the bar is designed as an Angular form, so I firstly installed the *@angular/forms* package, added an instance of the FormBuilder[[2]](#footnote-2) class in the parameter list of the component’s constructor, and then instantiated the form object. Like the search mode, the filter option makes a list of event types appear in the menu, representing a checkbox Angular form where the user can select multiple filters. When the submit button is pressed, in both cases, the form data is saved, used in an API call (either the one which gets an event by name or the one that gets events by type), and causes a response kept in an Angular service. Saving information directly in the map-option-menu component would have been an inappropriate approach since the searched or filtered events must be displayed on the map, so I used the service to connect this component to the one with the map functionality.

## **Historical Routes**

The routes mode of the application is one feature that makes the project different from other existing platforms for discovering History. When this mode is selected from the menu on the exploration page, a new map with a different design is loaded, and the events in the current timeline interval are placed as individual markers at their coordinates. A visual representation of a historical route is visible in Figure 3.5.

In the section regarding the search and filter functionalities, only the first two modes of the map page are explained. But what happened when the user opts for the routes mode? In this case, there must be created communication with the map component, so one status variable in the events service is switched to mark the routes mode is on. Its value will be obtained in the map component by calling a get function also located in the service file. Because any change of the status variable must be intercepted by the map component all the time, this leads to asynchronous operations, so I used the Observable Angular interface to set and get the variable, a method to handle a variety of asynchronous changes. Therefore, the status variable is an Observable, to which the map component subscribes in order to get the last update value whenever it is changed in the service.

After the new map is displayed, the markers placed on it will have a new type of listeners, triggering the appearance of paths between events when the user clicks on one map point. Two important aspects of the implementation are worth mentioning here: the back-end API for constructing the succession of events forming a route, and the usage of the Google class Polyline for drawing it. Firstly, when an event is clicked, the RoutesAPI GET endpoint is called, which creates a list of events with the same type and category as the selected one. The final route, the response, consists of these events sorted in ascending order by their date. Back in the Angular application, there are created instances of the Polyline class which represent lines on the map between each pair of consecutive events in the fetched list. They are added between the event markers with a delay, for creating an appealing visual effect, and when the routes mode is switched off, they are cleared and disappear from the map.

Finally, each historical route will have an information section which can appear on the page, providing a small description of the category it belongs to, and a list of all the events with their dates in chronological order. This particularity was added because it gives more insight into the chosen route, and helps users identify the events that compose it more easily.

A map of the world with red points

Description automatically generated

*Figure 3.5 Historical route representation*

## **Clustering Algorithm**

In the section dedicated to the event markers, there are described the conditions when the clusters are displayed. Now, in this part, I will go through the actual method which groups events based on their space coordinates, thus moving to the back-end implementation of the ClusterEventsAPI.

One first aspect is that the method receives via the request parameters the events to be clustered. Their coordinates are extracted in a list that will become the information the clustering model will use to fit and determine the groups of events. The classification model is imported from the *sklearn.cluster* Python library and is of type KMeans, meaning that it uses an algorithm of classification for unlabeled data sets into groups based on the k-nearest neighbors of each instance [20]. It results in several cluster centers (here called centroids) which are the arithmetic means of all the other points in their clusters, and in group labels for all the other points. One event being put in a specific cluster means that it is closer to its centroid than to the ones of all the other clusters.

The values used for the model include the number of clusters (*n\_clusters*), as well as the number of times it is run with centroid seeds that differ (*n\_init*). For the first one, I chose two different numbers in the following manner: if the list of events to be clustered has a length smaller than 100, I considered that a number of 5 clusters will suffice since it is a small collection, while for larger lists I opted for 10 clusters. These values prevent the map from becoming too crowded, but still manage to create a balanced distribution of the cluster icons. It is important to add that there would not be cases when these values become too small for the quantity of events since larger lists will be sliced to a maximum length of 500. For the parameter *n\_init,* I chose the value 3, which behaves well when it comes to the time execution of the algorithm.

When the response is received by the client, it uses the method for adding cluster icons on the map, and the coordinates of the centroids will become the actual position of these icons. Each marker has a label showing the total number of events grouped around that centroid and is added a listener for displaying additional data about the group on mouse-over.

# **Users’ System and Actions**

## **4.1** **Authentication System**

In order to explain the system of users and the methods for authentication, I will firstly describe the way the information for HistoryMapper’s users is stored in the database.

Django includes a predefined authentication system which completes the database with standard tables for users, tokens, user groups, and permissions. Since HistoryMapper does not include user groups and the permissions are handled at the Angular level, only the AUTH\_USER and AUTH\_USER\_TOKEN tables were used in the code. For storing passwords in the table for users, Django uses the PBKDF2 algorithm with a SHA256 hash, a method for deriving the cryptographic key, resistant to operations such as dictionary attacks [21]. Further information about this algorithm can be found in the standard Password-Based Cryptography Specification [22].

The user model is found in the *django.contrib.auth.models[[3]](#footnote-3)* package, which comes with the fields for the username, email, and password, and, apart from them, with some util methods that handle the authentication status and the values of the fields. However, there was an issue with this model which could not be handled into the Python code because of the difference in field types and data conversions, so I introduced in my application a Serializer class called User. Django Serializers are classes that transform the fields of another one in Python types and make the data easy to be converted into formats like JSON, which was useful for the logic of the authentication APIs [23].

At the level of authentication logic, two main methods are implemented: the registration and the login, but they are accompanied by other functions that make the operations

complete: the ones for user CRUD actions. Basically, the registration endpoint is of type POST because it adds the new user included in the request into the database if it is not previously registered and creates a token for it. On the other hand, the login endpoint is a PUT one, checking the existence of the username, the match between password and user, and adding a token in case the user misses one due to changes or expiration.

Turning to the Angular side, the logic is split into a page for each authentication step: the registration, the login, and the profile page, while the logout is a simple call of a TypeScript method. Both the sign-up and the sign-in are executed by constructing a request based on some values from page forms. When the application’s user wants to register, they need to introduce their data into the sections appearing on the interface, the request is created and sent to the API which adds the data into the storage system and returns a status response. In a similar manner, the login information is passed to be validated to the correspondent API endpoint. The most common exceptions (such as mismatch between username and password at sign-in or already existing user at sign-up) are handled via alert messages. Moreover, the forms in both pages contain validation for the fields, preventing the client from sending invalid information to the server, either a wrongly formatted email, a weak password or empty fields.

One other page is the profile one, having a separate component in the Angular structure. From this section, the authenticated user can check their saved information and edit them. It is a private page, together with the pages for quizzes and the admin section, meaning that, when the page loads, there must firstly be checked whether a user is connected to the application. Otherwise, the page redirects to a public one via Angular’s routing method: a private instance of the Router is passed to the constructor and the redirection is performed with the help of its *navigate* method [24].

But the logic of the private pages does not only involve a routing system. The core of the implementation here are some essential methods and data shared between components through a service. One field is the one storing the current user, that must be known to all the components that relate to the private features of HistoryMapper. Therefore, all of them have the user service injected into their constructor, which grants them access to the getter and setter methods. The setter adds the current user to the local storage for allowing its information to be kept when the page refreshes or reloads, while the getter obtains the current user from the local storage in each private component, which will then verify if this data is null or not. Nevertheless, the logout method, implemented in the service as well, simply deletes the user data from the local storage.

In Figure 4.1, I added all the actions the users can perform in the application based on their role, for having a clearer understanding of their permissions. It can be noticed that the unauthenticated user does not have access to any private page, but they can change their role by creating an account. The operations for users registered in the system are mainly related to the quizzes and the management of their profile, functionalities that can be accessed by the admin as well. Apart from them, the superuser of the platform has access to the database table storing events directly from the Angular interface, being able to perform a complete set of CRUD operations for it.

A diagram of a diagram with Turning Torso in the background

Description automatically generated

*Figure 4.1 Use cases diagram for all types of users*

## **4.2** **Admin Actions**

Even if the admin has full access to the CRUD methods on all the database entities through the Django interface [25], I wanted to implement these actions for events from the Angular application as well, creating a quicker navigation and data handling for the administrator of the platform. The activity diagram depicting the actions the admin can opt for from their private page is shown in Figure 4.2.

A diagram of a website

Description automatically generated

*Figure 4.2 Activity diagram for admin actions*

All the logic for the special admin actions is present on the dedicated page, accessed from the header menu. Not only is the icon for this page hidden when the admin is not authenticated, but the page is also made private, so it automatically redirects to the main page if the check of the admin authentication fails. For this verification, I chose a user that will be given the admin permissions by making its *is\_superuser* field in the database true and, each time this page loads, the component checks whether the current user has the required permissions.

The section comes with a menu, for selecting the desired CRUD operation: adding data from DBPedia automatically or visualizing events: the complete list or a particular event. The structure is the following: the operations of showing or deleting one event only require the name of the database entry, so no additional information is sent to the server. This is why they do not include any form on the interface to take data from, unlike create or edit, that need values to be added or updated. The edit and delete actions depend on a selected event, therefore they do not have a separate section in the menu, and simply appear as possible operations in the section for showing events.

I chose to perform the show operation as a method which gets an event by name rather by an ID because I already had this method implemented for the search engine. This is made possible due to the fact that the names of the events have a unique constraint in the database, thus the method cannot return multiple instances, which would have been an issue for the implementation logic. The edit and delete endpoints were then added under the same API class, and only the type of method and the request form differentiates them.

The other left methods are indexing the whole list of events and creating new entries. These ones are handled by separate APIs because do not depend on the name of a particular event. While the first one involves a GET method which simply obtains all the facts from the database, the second one involves the methods for obtaining entries from DBPedia and saving them in the storage system. This operation cannot be performed in the interface provided by Django, where only manual insertions are allowed. For specifying which data will be extracted from Wikipedia and added, the admin has sections in the form for the type of events, the category of DBPedia, and the new category, the one which will be present in the local database. The completion of the events sent to the server is checked by form validations, and any possible errors are handled with special messages.

A screenshot of a computer screen

Description automatically generated

*Figure 4.3 Admin section for adding events from DBPedia*

## **4.3** **Quizzes**

The system of quizzes belongs to the private part of the application, so only authenticated users can access them. They are created dynamically, classified by category, and depending on the list of events included in that category. So, there will be no entity storing quiz questions, the only one table related to the quizzes being the one which keeps the results of the users, the QuizHistory table.

I will firstly break into the logic for displaying the quizzes. After the user logs in, they can access the main page for quizzes, where separate sections for each category are displayed, along with the last score of the user for each of them, if the result exists. When a not signed-in user tries to access this page, they are automatically redirected to the login section. The way the quizzes page operates is by calling an API to obtain all the categories of the events in the database, and each group of quiz questions will be related to the historical facts included in one of these categories. Moreover, the last score for each quiz is fetched from the server in a similar way. Like all the other components who perform operations related to the database and need to be linked to other pages, the current one has a service injected in its constructor. It is in this service where the APIs for getting categories and quiz results are performed, and, in case the user selects a specific quiz, where the ID of the current category is set. What happens after the user clicks on one quiz section is that they are redirected to the questions page where the saved category ID is obtained from the service, and based on its value, a specific set of questions appears. The right answers are stored in a list, compared to the actual answers selected by the user in a form, and the result is calculated as a percentage value (the number of correct answers divided by the total number of questions and multiplied by 100). This value will be displayed in the results section after the user has submitted their answers. Also, it will be added into the QuizHistory table for the specific entry; the one linking the ID of the user to the ID of the category; via a PUT API endpoint, that either adds a new database entry or modifies the existing one, based on the current case. When the quizzes page is reloaded, the new result for this quiz will appear in its dedicated section, as shown in Figure 4.4.

A screenshot of a game

Description automatically generated

*Figure 4.4 Section of quizzes with associated user results*

Another important part is the actual creation of quiz questions. It does not add data in any table in the database because, each time the user selects a specific quiz, the questions could be different. The currently selected category ID is passed to the server method, and a random list of at most ten events included in this group is obtained from the storing system. After that, for each of these elements in the list, there is created a question of a randomly selected type from the following: 1 - guess the event name question, 2 - guess the location, 3 – guess the date. This type, along with the current event, is sent to another method that generates the question with its answers. The first two types of questions behave similarly, meaning that the correct answer will always be the name or the location of the current event, while the other answers will become three names randomly selected from other events in the database. In case of the last type, the guess-the-date question, the correct answer is the date of the event, while the wrong choices will be random years in the interval delimited by the current year minus 100 and the current year plus 100. The list of answers along with the index of the correct one are always retrieved by this method, converted to the JSON response of the API, and sent to the client. As it can be noticed, the logic involves creation of several random values, which was able with the use of the *random* Python library. Thanks to this import, the code is able to incorporate creation of pseud-random numbers, a logic which depends on the basic *random* function for obtaining an aleatory float uniform value in range [0.0, 1.0). All the methods which generate values in this module are dependent on an instance of the *random.Random* class. Some of the functions in this library that were useful for the creation of the quizzes were *randint* (for integer values), *sample* (for filling a list with random values) [26].

# **Technologies Used**

The HistoryMapper platform is composed of two applications linked together via API calls: the Django back-end part that is also connected to the SQL database, and the Angular one for the front-end part. Both sides interact with some external APIs as well, the Angular application for requesting Google maps, and the Django one for internal logic of its local methods. The schematic structure of Historymapper’s architecture is depicted in the component diagram in Figure 5.1.

A diagram of a computer network

Description automatically generated

*Figure 5.1 Component Diagram of HistoryMapper*

## **5.1** **Back-end**

Django is a Python framework for designing web applications, coming with a system that correlates models with templates, and views. I opted for its usage in my project because it involves an organized structure, is versatile, and I consider that coding

in Python is facile and fast. The project uses the Python 3.10.7 version, and Django 4.2.6. For the initial setup, I created the application called *explore* with the following command: *django-admin startproject explore.* The site can be started with *python manage.py runserver and* is running at port 8000.

The main layers of a Django application are the models, the views, and the templates. In the following paragraphs, I will explain the first two ones since the templates are generally responsible for presenting the information to the users and this part is handled, in the current case, by a separate front-end application.

The models describe the physical entities in the database and can also create them. They are designed as classes derived from the *Models.model* class, with fields for each attribute. The data types and the constraints such as the ones for primary key, foreign key or unique attributes are also created in the models [27]. The convention is that the names of the tables, apart from the ones related to the authentication, be in the following format: *nameOfTheApplication\_nameOfTheTable*. The back-end folder is called "explore”, and this is why all the physical tables (excepting the ones for users’ system) have the *explore\_* prefix in the database. Because all the changes in the models must be visible in the database, after any modification, I created a new migration (*python manage.py makemigrations* and *python manage.py migrate*) in the Django dedicated folder. It is important to specify that Django includes an ORM (Object-Relational Mapping), and this is why database operations can be simply executed in the Python code. However, manual actions are permitted as well, thanks to the Django interface for editing the tables mentioned in the *admin.py* folder.

The views are the code sections for the methods, in this case Django REST API endpoints for communication. All these functions are class methods, each API class inheriting the APIView class from *rest\_framework.views*. Other libraries used for this part are the ones that handle JSON responses (JsonParser from *rest\_framework.parser* and JsonResponse from *django.http*). Treating the status of a HTTP request is done via the *rest\_framework* status library. Considering the fact that the requests are passed via URLs, each API class must be mapped to a route, all defined in the *urls.py* file.

The last important file of the Django structure is the *settings.py*, where there is established the connection with the database, the installed apps, and other information such as the secret keys used in the application.

## **5.2** **Front-end**

Angular is a framework responsible for creating Single-Page Applications, where the content is displayed dynamically, without the need of reloading the HTML pages when changes occur. Its first version was released in 2010 as an open-source project, based on JavaScript language, and later emerged into a complex framework for building large-scale applications. The current version uses now TypeScript for the logic of the pages and the user interactions, and HTML with CSS for the styling [28]. The used versions of all the packages needed by the front-end application are displayed in Figure 5.2.

The front-end structure is divided into components, each being assigned a separate portion of the application’s logic with a different TypeScript, HTML, and CSS file, connected between them and added to the same folder. Because they describe different pages of the application, each of them has assigned its own route where it can be accessed. All of the components are mentioned in the *app.module.ts* file, where the architecture of the application is described, and their routes are located in the *app.routing-module.ts*.

One powerful concept of Angular is the Dependency Injection. It facilitates the interaction between dependency consumers and dependency providers, such as the components and the services for calling the back-end APIs [29]. But it is not the only useful element in this framework that contributed to the implementation of HistoryMapper. The Angular Router that establishes navigation from one page to the other was also included, along with some other concepts such as the Angular Forms for creating all the forms present in the application, the ngx-slider for the timeline axis (*@angular/ngx-slider*) or the HttpClient for the API calls (*@angular/common/http*). Another import which was necessary was the one of OnInit from *@angular/core*, that allows components to define their own ngOnInit method for handling behavior at the time of page load. Again, from the *@angular/core* package, the ViewEncapsulation is imported because I needed to control child encapsulation for the timeline axis, and not make it inherit the design from its parent.

The Angular part of the project communicates with the Django one via the API calls, and is hosted on a separate port, localhost:4200.

A screenshot of a computer program

Description automatically generated

*Figure 5.2 Versions of the technologies used in front-end*

## **5.3** **Database**

Oracle storage system is a relational database, meaning that the data is kept in tables which are organized in a logic of rows and columns, and the entries are. Also, the tables are interconnected via refence keys which point to primary keys (ID’s) of other entities. There are more reasons why I chose this type of database for my project, from the fact that it is compatible with many technologies, including the Django framework, to the one that Oracle allows users to easily install local databases on their computer with the help of their products. The manipulation language used is SQL Oracle. As previously mentioned, the storage system of HistoryMapper is a local Oracle Database, created with the installation of the Express 21c product, Version 21.3.0.0.0. For using it, I first created a user and granted all the permissions for it from the SQL\*Plus terminal. The credentials for the user, along with the name (XE), host (localhost), and port (1521) where the database is hosted are all required for establishing the connection in the *settings.py* file of the Django application. Connecting the database to the back-end is done through *oracledb* driver [30], and it was one of the first setup steps for the development process, right after the creation of the Django application.

# **Conclusions**

## **6.1** **Summary**

The development process resulted in a web application that brings learning History closer to modern technologies. I believe that what makes HistoryMapper powerful in the process of educational exploration is providing a complete spatial and temporal context for the historical data in a manner which is approachable by a wide range of users. To achieve that, I insisted on the interactive particularity of the application, as well as on the suggestive interface.

Achieving interaction with the user led to the necessity of a wide range of actions which can be performed related to the map, or to the navigation through historical periods and quizzes. The map options’ menu is one powerful section of the platform, because it gives indirect access to the operations performed in the back-end. In fact, many of the options available in the interface are related to API calls that access the storing system because it is one of the most valuable parts of the application, and mandatory for the existence of dynamic actions, such as the automatic transformation of the maps or the creation of random quiz questions. In this way, the platform can become a useful tool for teachers, who could present History in an appealing manner and have access to organized data, but it could also be appealing to historians for fast access to information, or, in general, for any person with a passion for History.

Regarding user interface of the final product, it highlights the suggestive particularity of the platform through an attractive display, including a header menu which is facile to navigate, and contains indicative icons, a home page menu, and a dedicated section for choosing interactions with the map: filtering, searching, showing historical paths. Apart from this nuance, I intended to provide an appealing design, choosing colors, markers and images that could be associated with the historical theme of the application. This is one reason why, for example, each type of marker has a characteristic icon image. Moreover, the two different designs of the maps are chosen to be chromatically integrated into the application, and to avoid monotony in the navigation flow. One appealing characteristic of HistoryMapper can be found

at the page of historical periods, whose images and display modalities are intended to give users the impression of a journey in time, along with the correlation to the map, and the timeline axis.

The potential of further development is a wide one, leaving space for constant improvement and change aligned with the general purpose of the application. An enriched user experience, actuality of the data and the technologies used, or enhanced interactions with the map are some of the main directions which could be followed to keep HistoryMapper an useful and well-structured platform for exploration of History.

## **6.2** **Further Development Possibilities**

One opportunity for further developing the application is adding more events to the database by creating automatic algorithms for extracting data from various sources similar to DBPedia database, and the logic could be expanded to the historical facts of the other continents, not only Europe. In the context of the constant occurrence of new events in Human History, this would also keep the application up to date with the newest historical facts, so data from the most recent years should be added. One benefit it could bring the platform is a more diverse system of quizzes, with the occurrence of a wider range of categories, and this could lead to some other types of quiz questions (for example, about some famous historical characters or dynasties).

Another direction which could be further exploited is the map functionality. It could include highlights of the areas with a large density of historical activity, personalized for each period, providing more insights into the dynamic of Human History. Moreover, the routes could be classified into specific domains, and even characterized by some personal description sections, including images or videos, like the information section for the events. I consider this would enhance the educational feature of HistoryMapper, giving users a wider perspective over the succession of events, and not only the individual facts themselves.

I would also add that, in the context of rapid development of Artificial Intelligence, one such model could be integrated into the application, for an advanced search system of the events and their context. Whether it is based on the natural language processing of written or verbal user commands, or the recognition of images uploaded by users to find out what event they are depicting, this would strengthen the interactive particularity, and make the discovery process easier and faster.

# **Bibliography**

1. *OpenHistoricalMap*, URL: <https://www.openhistoricalmap.org/#map=5/47.055/7.734&layers=O&date=1924-05-26&daterange=1824-01-01,2024-12-3>. Last accessed: 24 May 2024
2. *HistoryPin*, URL: <https://www.historypin.org/en/>. Last accessed: 24 May 2024
3. *TimeStamps*, URL: <https://timemaps.com/>. Last accessed: 24 May 2024
4. *Django - Documentation*, URL: <https://docs.djangoproject.com/en/5.0/>. Last accessed: 23 May 2024
5. *DrawSQL*, URL: <https://drawsql.app/>. Last accessed: 27 May 2024
6. *Rest API Introduction*, August 2023, URL: <https://www.geeksforgeeks.org/rest-api-introduction/>. Last accessed: 10 May 2024
7. Bobby Stearman, [*did\_django\_google\_maps\_api*](https://github.com/bobby-didcoding/did_django_google_maps_api), May 18, 2021, URL: <https://github.com/bobby-didcoding/did_django_google_maps_api>. Last accessed: 22 January 2024
8. WebbyFan, *How To Create Google API KEY***,** December 2018,URL: <https://www.bing.com/videos/riverview/relatedvideo?&q=how+to+obtain+google+api+key&qpvt=how+to+obtain+google+api+key&mid=B0758D27E6D656B74F39B0758D27E6D656B74F39&&FORM=VRDGAR>. Last accessed: 8 January 2024
9. *Google Cloud Platform*, URL: <https://console.cloud.google.com/>. Last accessed: 20 May 2024
10. *Google Maps Platform* - *Maps JavaScript API*, URL: <https://developers.google.com/maps/documentation/javascript/overview>. Last accessed: 24 May 2024
11. *Google Maps Platform – Geocoding API,* URL:<https://developers.google.com/maps/documentation/geocoding/overview>. Last accessed: 12 April 2024
12. *YouTube – Data API,* URL: <https://developers.google.com/youtube/v3>. Last accessed: 2 May 2024
13. [Jennifer Gunner,](https://www.yourdictionary.com/member/2299)  *Historical Eras: List of Major Time Periods in History* in *YourDictionary,* July 2022, URL: <https://www.yourdictionary.com/articles/historical-eras-list>. Last accessed: 20 February 2024
14. Adam Hart-Davis, History, *The Definitive Visual Guide,* Publishing: *Dorling Kindersley Limited,* London 2007

*Middle Ages* in *History*, Updated June 2023, URL: <https://www.history.com/topics/middle-ages/middle-ages>. Last accessed: 20 February 2024

*English History Timeline* in *OnThisDay,* URL: <https://www.onthisday.com/countries/united-kingdom/england>. Last accessed: 20 February 2024

1. *About DBPedia,* URL:[*https://dbpedia.org/page/DBpedia*](https://dbpedia.org/page/DBpedia)*.* Last accessed:16 May 2024
2. *SPARQL 1.1 Query Language,* March 2013, URL: <https://www.w3.org/TR/sparql11-query/>.Last accessed: 10 May 2024
3. *Google Maps Platform – Advanced Markers,* URL: <https://developers.google.com/maps/documentation/javascript/reference/advanced-markers>. Last accessed: 12 May 2024
4. *Maps JavaScript API – Simple Polylines, URL:* [*https://developers.google.com/maps/documentation/javascript/examples/polyline-simple*](https://developers.google.com/maps/documentation/javascript/examples/polyline-simple)*.* Last accessed: 12 May 2024
5. Cory Rylan, *Creating a Dynamic Checkbox List in Angular*, May 2018, URL: <https://coryrylan.com/blog/creating-a-dynamic-checkbox-list-in-angular>. Last accessed: 22 march 2024
6. *K-Means Clustering – Introduction*, March 2024, URL: <https://www.geeksforgeeks.org/k-means-clustering-introduction/>. Last accessed: 24 April 2024
7. *Practical Cryptography for Developers*, URL: <https://cryptobook.nakov.com/mac-and-key-derivation/pbkdf2>. Last accessed: 20 May 2024
8. B. Kaliski, *Password-Based Cryptography Specification,* September 2000, URL: <https://www.ietf.org/rfc/rfc2898.txt>. Last accessed: 20 May 2024
9. *Django REST Framework – Serializers,* URL: <https://www.django-rest-framework.org/api-guide/serializers/>.Last accessed:15 April 2024
10. *Angular Archive – Routing and Navigation,* URL: [*https://v2.angular.io/docs/ts/latest/guide/router.html*](https://v2.angular.io/docs/ts/latest/guide/router.html)*.* Last accessed: 24 May 2024
11. *Django Documentation - The Django Admin Site,* URL: <https://docs.djangoproject.com/en/5.0/ref/contrib/admin/>.Last accessed:7 March 2024
12. *Python 3.7.3 Documentation, Generate pseudo-random numbers,* URL: <https://documentation.help/python-3-7-3/random.html>. Last accessed: 24 May 2024
13. *Django Documentation – Models,* URL: <https://docs.djangoproject.com/en/4.2/topics/db/models/>.Last accessed: 25 May 2024
14. Dave Gavigan*, The History of Angular,* April 2018, URL: [*https://medium.com/the-startup-lab-blog/the-history-of-angular-3e36f7e828c7*](https://medium.com/the-startup-lab-blog/the-history-of-angular-3e36f7e828c7)*.* Last accessed: 3 June 2024
15. *Angular - Understanding dependency Injection,* URL: <https://angular.dev/guide/di/dependency-injection>. Last accessed: 25 May 2024
16. *Python-oracledb,* May 2024,URL: [*https://pypi.org/project/oracledb/*](https://pypi.org/project/oracledb/)*.* Last accessed:27 May 2024

1. Django creates more tables for authentication system, but only the used ones are drawn in the schema. [↑](#footnote-ref-1)
2. https://angular.io/api/forms/FormBuilder [↑](#footnote-ref-2)
3. <https://docs.djangoproject.com/en/5.0/ref/contrib/auth/> [↑](#footnote-ref-3)