SoulMind: Interactive Platform for Domain-Specific Data Analysis and Visualization

Alpár Cseke
Babeș-Bolyai University
Cluj-Napoca, Romania
csekealpar@gmail.com

István Király
Babeș-Bolyai University
Cluj-Napoca, Romania
kiraly_istvan1@yahoo.com

Károly Simon
Babeș-Bolyai University
Cluj-Napoca, Romania
ksimon@cs.ubbcluj.ro

Szilárd-Gábor Mátis
Codespring
Cluj-Napoca, Romania
mszg92@gmail.com

Kincső Tüzes-Bölöni
Codespring
Cluj-Napoca, Romania
tuzes-boloni.kincso@codespring.ro

***Abstract*—The purpose of the application to be presented is managing personal and historical data related to priests and the parishes they served at. The website offers a comfortable platform for recording new data or modifying preexisting entries. By not only providing a wide range of filtering tools and presentation options, but also a clear browsing experience, it aims to help in revealing correlations in the dataset, encouraging and helping research in the field.**

The SoulMind platform also features a role-based multi-tier user management system where new users must request access via a registration form evaluated by the administrators of the website. This is in order to protect the personal information stored in the system and fine-tune its accessibility.

The paper presents the functionalities and the usage of the application, introduces the reader to some implementation details and provides a brief overview of the used tools and technologies.

I. Introduction

There are a number of thoroughly customizable CRUD platforms (Create, Read, Update, Delete - the four essential operations on data items) already available. Therefore it may seem as there is no demand for new applications with the objective of managing a collection of entities belonging to a specific domain, but there is an aspect with limited support within the available platforms. When dealing with Bad Data [1], end products generally settle on underdeveloped features.

The software to be presented in this paper offers a solution for one instance of this problem. It is a web platform developed for the management of a still growing, rather valuable dataset consisting of various information about the Protestant priests of Transylvania from 1800 up to the present days. This dataset exhibits multiple characteristics of Bad Data, such as missing fields, improper formats, inappropriate data (entered in the wrong fields), duplicate and/or misspelled entries. Due to the sheer size of the data (more than a thousand entries, each with more than a hundred different fields), manually cleaning it was not an option.

Aside from making these records easily accessible and extendable, there is a second key motivation behind the project. The dataset offers unique information about its subject, making it a valuable basis for different types of researches. The SoulMind project aims to provide two different approaches for this, both reliant upon an extensive data filtering module: bibliographical research by searching for individual cleric profiles and statistical research assisted by cumulative filtering and displaying options.

There are two additional rather important requirements of the application. First, the personal character of the data necessitates secure access, which is accomplished by a multi-tier user management system, where even the lowest tier is evaluation-based and higher ranks offer wider access and more functionalities. Second, because the target audience of the website covers technologically less knowledgeable people, the user interface was designed based on multiple UX (User Experience) [2] research methods.

II. Functionalities and user roles

Since the managed data has a very personal nature, access to it is broken down into multiple user roles. The section presents the functionalities of the software platform, grouped by their required minimum access level.

1) Visitors of the page: Users who are not signed in can access only the main page containing the news stream and the description of the project. The authentication operations include the login, the forgotten password and the access request forms. The latter only initiates the registration process, the account's creation depends on administrator approval.

2) Guest role: Logged in users with the Guest role are able to access the searching pages and the chart creator too. When searching priests, only the data shown on the result cards is available, Guests cannot navigate to the priest profile pages. However, they can browse the parish profiles with all existing data, which is accessible from the placement list of priests or from the dedicated parish searching components. The latter is available both in list view and on a map interface.

3) Reader role: Reader or higher ranked users can benefit from one additional functionality, the priest profile pages, where all available personal data is present. The page is reachable from the parish profiles and naturally from the priest filtering page too.

4) Editor role: The role of Editors cover two more feature groups. They can create new data sheets for priests or parishes by providing some basic information about them, if these do not exist yet; in case of matching data, they are redirected to the already existing entity. On the profile pages, the edit button will appear for Editors, making it possible to change fields and manage files and pictures. On the priest profiles, new family members, placements, qualifications, occupations etc. can be added as well.

5) Admin role: The Admin is the highest user role. Aside from all previous functionalities, they are granted access to all user data (name, e-mail, role), and they are able to delete existing accounts. However, their most important task is to evaluate registration requests, which also takes place on the admin dashboard. Admins can decide to accept or refuse new accounts, and the user's rank is also determined by them. They can also include the reasoning behind their decision, which will be part of the notification e-mail sent to the user. Last, but not least, only these users can create, edit or delete news on the main page.

6) Owner role: There is also an Owner rank. Its access rights are the same as the Admins', but there is only one such account and it cannot be deleted, contrary to Admins.

III. Architecture overview

A. Overall architecture

The software platform's architecture can be divided into three well separated components. The bottom layer, a document-oriented MongoDB database, is responsible for the permanent data storage. This gets queried by a backend server implemented in Golang, where the retrieved documents are first encoded into intact models, then these are further transformed into more compact transfer objects specific to the request. These requests are received at the server through a RESTful API [3], and are originating from the web client implemented in TypeScript, built on the React web framework.

The overall architecture is based on the Model-View-Controller (MVC) design pattern. With the three-layered approach's view part being implemented in React, the application can be also categorized as a Single Page Application (SPA), meaning that the view layer is fully built up in the frontend layer [4]. Even the business logic resides predominantly here, while the backend server's only tasks are about providing and managing data.

The application is deployed in a multi-container environment. The three components are communicating via an internal network, while a fourth container - incorporating a proxy server - exposes the platform to the outside world and distributes requests between the web client (for static files) and the backend server (for data access and manipulation).

B. Backend server architecture

The data access layer of the server is organized around general interfaces. Currently, there is a MongoDB implementation for these, but the possibility for extension or replacement is given, because other layers do not contain any code specific to the database.

Due to the frequent reinitialization of the production database (see Subsection IV-A), the backend server also communicates with a MySQL database. These importing functions are only ran at server startup and they batch process the data. This database is not part of the end product, it was only used in development.

Data transfer between the repository and controller layers is realized by the internal entities of the model package, while the RESTful API's incoming and outgoing objects belong to the DTO package. The usage of Data Transfer Objects provides multiple benefits, among them are the reduction of message sizes and quantities, the exclusion of unauthorized data and the possibility of type conversions. They also streamline the development of the web client by preselecting the required data.

The detailed architecture is also visualized on the Fig. 1 component diagram.

C. React state management without a store

A distinct feature that determines the web client's architecture is the inclusion of the recently released Hooks API. Hooks are functions by which a component can attach to the component lifecycle. They offer solutions for both of React's biggest shortcomings. Classes are replaced by functional components, thus events related to the same element are not scattered in different lifecycle methods, but are grouped together by the *useEffect* hook which triggers on data change. State management is also simplified, global store-based libraries like *Redux* or *MobX* are replaceable by *useState* or *useReducer* calls [5] operating on immutable variables via pure functions. The usage of these hooks makes the architecture of the frontend component inherently simple, free of additional layers.

Developers are also able to create their own hooks, making special logic reusable between multiple components. In the project, among other uses, custom hooks help in structuring the state for the asynchronous HTTP requests' variables, such as response, error and the loading's phase.

IV. Implementation details

This section explains some unique problems and their solution, granting an in-depth look into the development of some functionalities and requirements of the project.

A. Steps of the data processing

The provided database was initially in Access, and it was very poorly maintained. It exhibited significant differences between the priest records regarding the available data quantity and quality.

Due to this, the data had to be transformed into a more usable format. The first step was converting the single-table Access file into a relational SQL database, in order to categorize the data and better understand how it is built up. Then the *Priests* and *Parishes* MongoDB collections were built up from these tables.

The original database only contained explicit data about priests, the collection of parishes was generated by an algorithm that aggregated data from priests' placements. Due to the misspells present in the original database, this algorithmic data propagation created multiple parishes with very similar names. To prevent potential data loss, most of these could not be algorithmically cleaned by the team, since many village names only differ in one character from one another, bigger towns often accommodate multiple parishes and some of the original data fields contain additional information about the

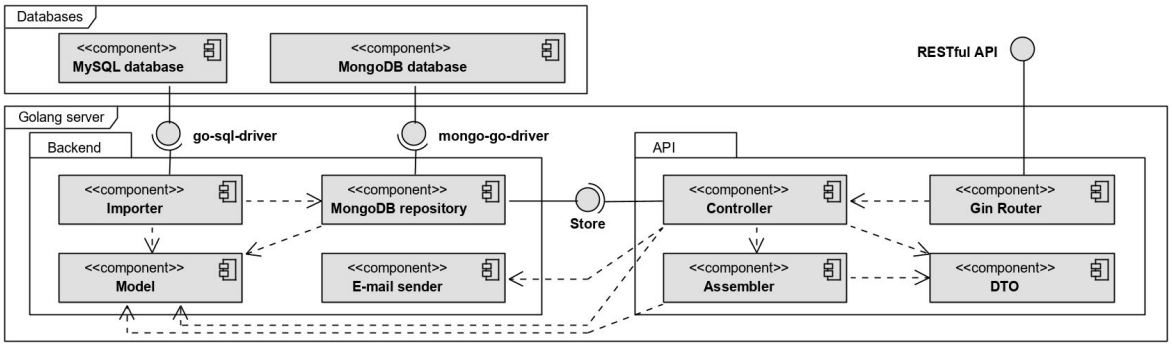

Fig. 1. Components of the backend server and their interactions.

priest. Ultimately, the merging of the duplicate parishes and the clarification of the priests' biographical data will be done by the Protestant institute's librarians.

However, most of the parishes have coordinates, which were retrieved and inserted by two Python scripts querying their names using Google's Places API with a location bias favoring hits close to Transylvania.

B. Displaying and managing dates

The correct management of dates was essential, since the searching components heavily rely on them. In the original database these were saved as strings, and contained a vast number of misspells, ambiguities and extra information. In order to normalize them, a new structure was created, which stores the year, month, day and original string values in different fields. The transformation of the data is performed by an algorithm exploiting multiple regular expressions, attempting to recognize as many components of the date as possible.

The presentation of these dates on the web client is implemented by two custom components, thus making the involved custom logic easily reusable across the website. The *DateInput* consists of four inputs for the four fields (one of them being the legacy string or a comment), and of a conditionally appearing error message. If any error is introduced while in editing mode, committing the changes is not possible.

The *DateDisplay* component aims to present the dates as precise as possible. If numeric values are present, a full or partial date is returned. When mousing over these, the original string value from the initial database is also shown in a popup text-bubble. If the year could not be retrieved, this original string is shown instead of the date. Finally, if that is also empty, a *No data* message appears.

C. Custom chart designer

The data visualization is implemented with the *Reaviz* package [7]. Its most prominent advantage, from the point of view of the project, is that it utilizes the same data model for all of its available charts. Therefore, data aggregation can be realized in a simple and concise manner, independent of the chart's type. Another advantage of the library is that it is opinionated, meaning that most of its design choices are made internally, so the theming's data-specific implementation is not falling on the developer.

This data processing is implemented on the frontend, with the combination of multiple functional operations such as *forEach*, *reduce*, *filter* and *sort*. The underlying data model received from the backend server only contains the ready to use values, i.e. years instead of full dates and only the sizes of different data sections. The data fields for the grouping and aggregate values are selected by the user and are dynamically accessed in the algorithm from these simplified entities.

The algorithm also features an optional unification of small groups, which improves the comprehensibility of the chart. Its threshold is also user configurable.

D. Automatic e-mail notifications

The application relies on Google's SMTP (Simple Mail Transfer Protocol) server for sending out notification e-mails to admins in case of a registration request, to new users after account approval or refusal, and when changing password. Custom data, such as user details, the password change token, and introduction or decision messages are dynamically inserted into the HTML e-mail templates.

Since depending on an external service always carries some uncertainty with itself, the server continuously monitors its situation. Whenever an error occurs in e-mail sending, the admin dashboard web page will show a notification, that the maintainers of the application should be contacted. This error disappears when new e-mails can be successfully sent again.

E. Security measures

The backend server and the frontend client both take multiple different precautions in order to realize data protection.

Through the use of DTOs, unauthorized clients cannot access any restricted data, not even directly from the API. This field-selection also encompasses leaving out internal identifiers and timestamps from the DTOs, whose transmission to the frontend client could cause potential security holes.

The backend's router contains a custom interceptor for filtering out unauthenticated senders. It can restrict access based on user roles too if additional parameters are provided. The navigation bar keeps the unavailable paths hidden, thus making certain functionalities inaccessible. Similarly, the data

manipulation controls are hidden too, when they are not applicable, such as the *Edit* button on the profiles or the article operations on the main page.

The navigation paths listed via the custom *GuardedRoute* wrapper function built around the *react-router* provided *Route* component takes this a step further. The function receives the variables necessary for the *Route*, namely the *path* and *render* values, but also the *user* object encapsulating the data of the logged in user and a list of permitted user roles. By the former two, it can be determined if the *Route* component should refer to the given path or to an error page. This way, unauthorized pages are not accessible in the possession of the URL leading to them.

Passwords are encrypted and salted multiple times. Effectively, they can never leave the application server, since they are not part of any outgoing DTO, and the proxy server communicating with the outside world is not connected to the database.

F. Responsive and clear UI

Creating an easy to use and intuitive user interface was one of the platform's key requirement. Multiple design plans for all components of the future website were created, the selection between these was made based on the remarks of the client and on further UX research.

The SoulMind platform does not have an Android or iOS application. It would have been only necessary if it relied on some native functionalities of smartphones, such as camera integration or location access. Instead, the user interface was designed in such a way that its arrangement fits any screen size from smartphones to widescreens.

The responsive interface and the underlying design was mostly realized using the React-integrated version of the *Semantic UI* (SUIR) package. The components' size, color and other visual attributes can all be configured through optional constructor parameters, but in multiple cases these were overridden by manually given CSS commands so they complement the overall look better.

There were multiple other methods for realizing the responsivity of the webpage. For example, the navigation bar has two implementations, one containing only the menu options' icons without the labels. Between rendering these two components the decision is made by SUIR's *computer/tablet/mobile only* selectors. Browsing on a tablet, the different cards of the search and profile pages automatically stack into one column instead of the usual two, by the *Grid* layout definer's *stackable* parameter. On mobile screens the main content is placed below the auxiliary controls otherwise present on the left side of the pages, while cards still show up in one column. The current resolution is taken into account using the *min-width* and *max-width* CSS media queries.

One essential difference between UX and UI (User Interface) design is that the former is not only concerned with the visual components of the page, but also considers the usability and the convenience of the functionalities. There are multiple small comfort features present in the web page, such as the reset buttons near the double sliders, the real-time search results, or the filter values' and page number's insertion into the current URL. The latter is implemented so that specific searches and results can be shared via the hyperlink.

To fine-tune the interface's font types and colors, the SUIR package was recompiled by using a library called *craco*, so that the internal configuration files of SUIR became accessible.

V. Technologies and tools

This chapter briefly introduces the reader to the technology stack of the application, followed by development methods and tools used in the project.

A. Server-side technologies

For permanent data storage a **MongoDB** database is used [8]. It is a NoSQL solution, which organizes the stored records into schema-less collections. These documents are syntactically similar to JSON. They consist of key-value pairs, where the value can be scalar, a list or even an inserted subdocument. Being document-oriented, it directly matched the object domain of the server, thus simplifying the queries.

The backend server is implemented in **Golang** [9], a performance and simplicity oriented language developed and maintained by Google. Golang is most often compared to the C language, since it does not feature classes, inheritance, generic types or even exception handling try-catch schematics. It is compiled, thus provides runtime performance and stricter syntax checking, but also features garbage collection.

The RESTful API was built with the **Gin** HTTP web framework, which pairs the incoming requests with their handler functions from the controller. It is enhanced by multiple middleware layers for session management, role-based access control and customized logging. For configuration management **Viper** is used, while the **mongo-go-driver** package acts as a bridge between the backend server and the database.

B. Client-side technologies

The platform's web interface is implemented in **TypeScript**, a typed and compiled superset of JavaScript [10], and is built using the **React** frontend framework [11]. React is component-oriented, meaning that the elements of the user interface are grouped and organized in components reusable across different parts of the application, and even in other web pages.

As for pre-created components, the react-compatible **Semantic UI** framework is utilized. The **Axios** library is responsible for sending HTTP requests to the backend server, relying on a promise-based asynchronous system.

Creating the map of the parishes was made possible by **Leaflet**, which retrieves the tiles required for assembling the currently displayed map from the OpenStreetMap API.

C. Testing, version control and deployment

The backend server's verification features both unit tests and extensive API testing in PostMan. The web client is inspected via snapshot testing, a technique reliant on mocks for isolating components and checking if their behaviour unexpectedly changes.

Version control is realized in Git. The git repositories hosted on GitLab are also connected to CI/CD (Continuous Integration and Continous Deployment) pipelines [6]. These automatically execute a list of interdependent tasks on the code pushed to the backend and frontend repositories, such as static code analysis (with golangci-lint and TSLint), running the tests and calculating the coverage, compiling the application and creating system images with the built components.

A third pipeline is responsible for the automated deployment. The Docker images created by the other two pipelines' last stages is accessed from GitLab's registry and the individual containers are connected together by the Docker Compose tool to be released in the test server.

VI. The Usage of the Platform

Before signing in with an account, only the application's homepage and the usual authenticating operations, such as registering (see Fig. 2), signing in or requesting a password change are available. The website is translated to Hungarian and English as well, and the selected language is saved between visits.

The homepage (see Fig. 3) contains a short description and a stream of news regarding the application and the community. After logging in, administrators are able to create, edit (see Fig. 4) or delete the articles. Similarly, all functionalities of the website are distributed across different user roles.

Among the menu items, in the *Search* drop-down menu one can choose to browse priests or parishes. Priests can be filtered and target-searched by a broad range of different controls present on the left side of the page (see Fig. 5). The paginated search results on the right side respond in real time to any filter changes. By clicking on the cards, the user is redirected to the priest profile (if the account has authorization). The names of the parishes are also clickable on these cards, leading to the parish's profile.

Searching parishes is also carried out by different filters: its name, location and its minimum number of priests. There are two ways to show the results, the first is similar to that of priests'. Aside from the list view, the data can also be displayed on a map as well (see Fig. 6). Here, parishes close to each other are clustered into a number showing their count, zooming in breaks these clusters down. Cards in the list and the marker pop-ups can be clicked to navigate to the associated parish profile.

The next selectable menu item takes the user to the chart creator page. On this page (see Fig. 7) a wide range of data visualizations can be easily created and customized through multiple parameters and the aforementioned filtering component. The most important of these options are the two drop-down menus where the displayed data fields can be selected. The first input specifies the main axis, the attribute by which the data will be grouped, such as birthdate or birthplace. In the second drop-down the value assigned to these aggregated categories is given. This so-called secondary axis can simply be the group's size, but also can take different averages such as the number of children or of placements. Furthermore, the user can choose the type of the figure: pie, bar or line chart. For the former two, three other options will be available for sorting the data and keeping the visualization clear (smoothing by aggregating or omitting small groups). Due to the combinability of the multiple options, currently 630 different charts can be generated based on a dataset.

There are several navigation links within the website leading to the profile pages. Both on priest and parish profiles, different tabs are available in the sidebar, to partition the data into categories such as general data, family, qualifications, placements, occupations, path of life, literary works, files, pictures

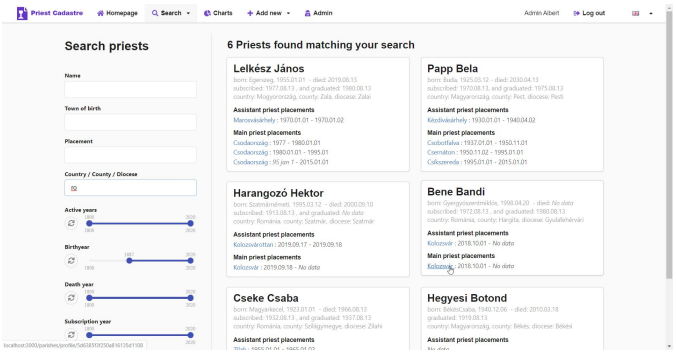

Fig. 5. Priest search page with filtering options and results.

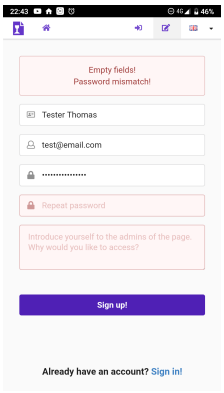

Fig. 2. Registration page with wrong input.

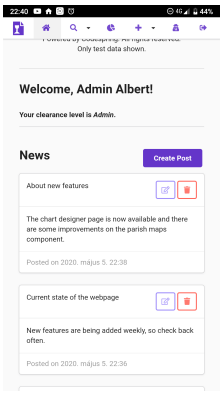

Fig. 3. The platform's homepage with articles.

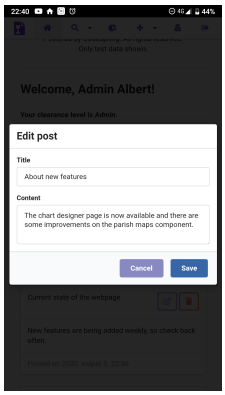

Fig. 4. Article editing by an administrator.

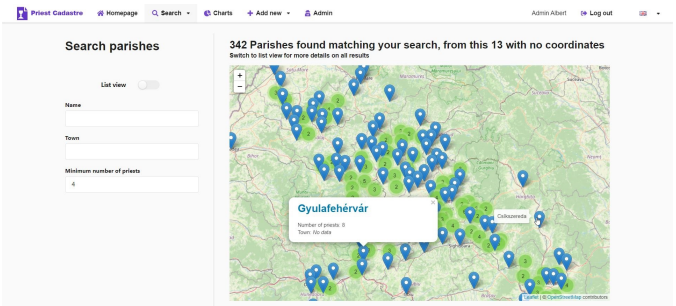

Fig. 6. Parish search page with map view.