**3 Creating “Cercenatorul3000”**

In this chapter, we dive into the practical aspects of creating an Application with a web scrapper and a recommendation engine, exploring the development journey of ”Cercenatorul3000,” an innovative platform designed to help Scouts all around Romania.

**3.1 Understanding the Context**

Organizatia Nationala "Cercetasii Romaniei" is the largest ONG in the country. Some of their core values ​​are personal development, community help, and teamwork learning. These values ​​are promoted through various activities, such as weekly meetings or weekend gatherings. Another method from the scouting methodology for developing the aforementioned values, the most favored one, ​​is organizing camps, multi-day events with a lot of activities and challenges. These camping experiences hold significant importance and popularity among Scouts, serving as vital opportunities for skill development, team building, and fostering a sense of camaraderie within the community. However, the organization of these camps heavily relies on the voluntary efforts of association members, who dedicate their time and energy without monetary compensation.

It's essential to recognize that these volunteer members often juggle their camp organization responsibilities alongside their primary jobs or other commitments. This dual workload can result in inefficiencies in camp planning and execution, as well as a lack of full engagement from organizers. The inherent limitations of volunteer-based organization can lead to errors, oversights, and even a sense of monotony in camp activities over time.

Here are some ways an application can revolutionize the organizing of camps:

Automating Information Retrieval with Web Scraping

One significant challenge faced by organizers is the time-consuming process of manually gathering essential details such as new and interesting routes for camps. Web scraping technology offers a solution by automating this process. Users can input their preferences, such as difficulty and duration, and instantly identify suitable and novel routes. By automating data collection and eliminating the need for manual searches, web scraping streamlines the planning process, allowing organizers to discover unique and enjoyable routes more efficiently. This not only saves time but also enhances the overall camping experience by introducing fresh and exciting paths that might not have been considered otherwise.

Enhancing Memory and Attention to Detail with Algorithmic Assistance

The complexity of organizing events like camps demands meticulous attention to various factors, especially when it comes to choosing locations, routes, terrain, counties, and cities. Human memory and attention to detail often fall short in effectively managing all these aspects, leading to oversights and inefficiencies. Advanced algorithms can assist in overcoming these limitations by taking all these location-related factors into consideration and generating a detailed and efficient plan tailored to the user's preferences. Additionally, databases populated with relevant data ensure the generation of optimal solutions based on historical data and user preferences. This approach not only streamlines the planning process but also enhances the overall experience by ensuring that all location-related details are thoroughly and accurately managed.

Fostering Creativity and Innovation with AI Recommendations

Recycling ideas and activities can diminish the novelty and excitement of events. To combat this, a recommendation engine can suggest new routes that users might not have considered, which can turn out to be much more enjoyable than expected. These recommendations aim to boost the authenticity and creativity of the experience, providing organizers with fresh ideas and innovative approaches to engage participants. By introducing unique routes and locations, the recommendation engine ensures that each camping event offers a distinct and memorable adventure for Scouts.

**3.2 Methodology**

The Methodology / Procedure section details the step-by-step approach taken to develop and implement the "Cercenatorul3000" application. It describes the specific methods and processes followed to achieve the goals of the project, ensuring that the reader understands how the application was built and how it functions.

**3.2.1 Requirement Analysis**

The first step in the development process involved a thorough analysis of the requirements for the "Cercenatorul3000" application. This included:

**3.2.1.1 Understanding User Needs**

To fully understand the needs of the scouts and to design an application that meets them effectively, I engaged in detailed discussions with a scout leader, who also serves as an experienced camp organizer. Together, we identified the fundamental requirements of our users for the "Cercenatorul3000" application. Our primary goal was to develop a user-friendly and intuitive UI, easy to navigate for users, guiding them through all the necessary steps to determine the location of a camp. For this purpose, we decided to implement a decision tree within the interface, where each question leads to a set of possible answers. Based on the user's responses, a node is selected, allowing the selection of other nodes that are subordinate to that main node. Additionally, we recognized the need for a robust and scalable database to store all projects and associated data, enabling users to select and manage the information required for camp planning efficiently and effectively. Additionally, we recognized the importance of leveraging advanced technologies such as web scraping and recommendation engines to enhance the user experience.

**3.2.1.2 Defining Functionalities**

Outlining the core functionalities needed, such as location selection, terrain preferences, hike options, and city choices, was a crucial step in the development process. Each functionality was carefully designed to streamline the camp planning process and ensure user satisfaction.

**The Login/Sign Up Page**

The login/sign-up page is the first screen users see when they open the application. This page allows the user to log in to the application or create a new account. Users enter their credentials to access the home page.

**The Home Page**

The home page is the initial screen users encounter after logging in. This page provides an overview of the application and serves as the gateway to all main functionalities. Here, users can create a new project or access the navigation bar.

**The Navigation Bar**

The navigation bar contains three key buttons:

Saved Projects: This button allows users to view, update, or delete their previously saved projects.

How It Works: This button opens a pop-up that explains how the application functions, providing users with a quick tutorial.

My Profile: This button takes users to their profile page, where they can view and edit their personal information. The navigation bar also displays the user's photo and email address.

**General information**

The first page you get when starting a new camp, where you can add important details such as name of the project, number of persons and number of days.

**Select a county**

The "Selecting a County" functionality serves as the cornerstone for planning Scout camps within the "Cercenatorul3000" application. This initial step empowers users to designate the administrative region where they intend to organize their camping adventure. By choosing a county, users delineate the geographic scope of their exploration, setting the stage for subsequent decisions regarding terrain, regions, hikes, and cities. With a comprehensive array of counties available, users can make informed decisions that align with their camping aspirations.

**Select a terrain**

The "Selecting a Terrain" functionality allows users to specify the type of landscape they prefer for their Scout camp. This step is important for tailoring the camp experience to the desired level of adventure and comfort. The application presents a range of terrain options based on the selected county, such as mountains, hills, forests, or plains, each with unique characteristics and challenges.

For instance, in Brasov County, users might choose mountainous terrain for a more rugged and adventurous camp experience. The application filters these options to ensure they are relevant to the chosen county, avoiding mismatches like offering delta terrains in a mountainous region. This targeted approach ensures that the terrain selection aligns with the environmental conditions and logistical capabilities of the camp.

**Select a hike**

The "Selecting a Hike" functionality empowers users to tailor their camping experience by choosing specific hiking trails within their selected region. This pivotal step ensures that the physical activities planned align with the skill levels and interests of the participants, contributing to a safe and enjoyable outdoor adventure. Users have the flexibility to customize their hike selection based on various criteria, including difficulty, duration, and region, filtered according to the chosen county. Leveraging web scraping technology, the application aggregates a diverse array of hiking trails from “Muntii Nostrii”, providing users with an extensive selection of options to explore. Additionally, users can benefit from recommended hikes generated by a recommendation engine, which considers user preferences and historical data to suggest trails that align with their interests and abilities.

**Select a city**

The "Selecting a City" feature within the "Cercenatorul3000" application is a crucial step in the camp planning process, enabling users to pinpoint the specific urban areas near their chosen camping region. This functionality ensures that logistical considerations, such as transportation accessibility and proximity to amenities, are carefully accounted for when finalizing the camp location. Users are presented with a curated list of cities within the selected county, each offering its own unique blend of accommodations, dining options, and recreational facilities.

**Generating the outcome**

The "Generating the Outcome" functionality in the "Cercenatorul3000" application is designed to consolidate all the information gathered from previous selections into a comprehensive and cohesive plan for the Scout camp. This feature ensures that organizers have a clear, detailed overview of all the crucial elements, allowing for efficient final preparations and execution.

**3.2.1.3 Technical Feasibility**

Evaluating the technical feasibility of implementing features like web scraping and AI-driven recommendations.

Web Scraping:

Objective:

The primary goal of web scraping is to extract up-to-date and relevant information about hiking trails and campsite locations from online sources, specifically "Muntii Nostrii".

Feasibility Evaluation:

Technology Stack: Python, utilizing the Beautiful Soup library.

Data Source: "Muntii Nostrii" website.

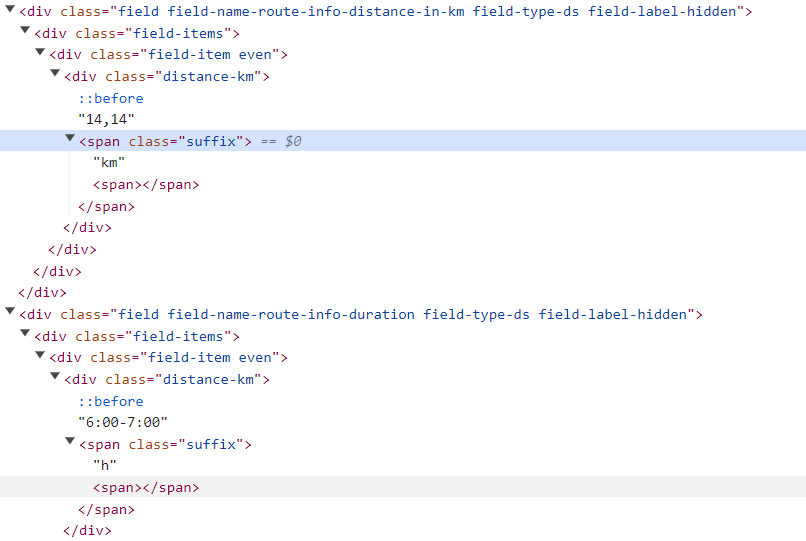
Data Extraction: Efficient scripts to extract trail names, difficulty levels, distances and duration

Data Storage: A scalable database schema using Firestore.

Challenges: Legal and ethical considerations, reliability of data, handling website structure changes.

Conclusion:

Web scraping from "Muntii Nostrii" is technically feasible using Python and Beautiful Soup, ensuring accurate and compliant data gathering.



AI-Driven Recommendations:

Objective:

To offer users personalized recommendations for hikes and campsites based on their preferences and historical data.

Feasibility Evaluation:

Technology Stack: Python, employing collaborative filtering.

Data Collection: User input data from previous projects, integrated with scraped data from "Muntii Nostrii".

Model Development: A collaborative filtering algorithm trained on historical data.

Personalization: Algorithms consider previsious projects and hikes input from the web scrapper.

System Integration: Integration with the application's UI for user-friendly presentation.

Challenges: Data privacy, model accuracy improvement.

Conclusion:

AI-driven recommendations, utilizing collaborative filtering in Python, are feasible, leveraging user input data and continuous learning.

Overall Feasibility:

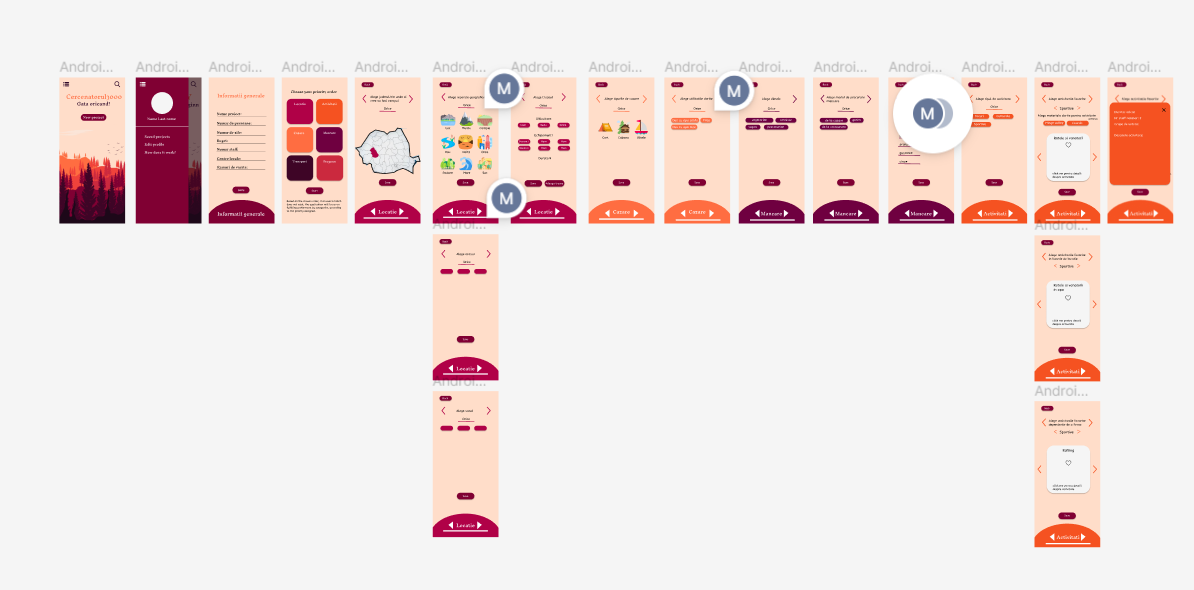
The integration of web scraping from "Muntii Nostrii" and AI-driven recommendations, both implemented in Python, ensures the "Cercenatorul3000" application can provide accurate and personalized guidance for scout camp planning.

**3.2.2 Design Phase**

Based on the requirement analysis, the next step was designing the application. This phase included:

**3.2.2.1 User Interface Design**

The first step in creating the "Cercenatorul3000" application was designing a user interface (UI) mockup using Figma. This was essential to visualize what users would see and how they would interact with the application's features. A coherent design is crucial for providing a quality user experience, especially for a product aimed at a large audience. Capturing users' attention and ensuring they have a satisfying experience is vital for the application's success. By carefully selecting a color palette and ensuring that sizes and images harmonize well, we created a visually appealing and intuitive design in Figma. This approach made it significantly easier to translate the design into code later on, ensuring consistency and ease of implementation.

****

**3.2.2.2 System Architecture Design:**

Developing a high-level architecture for the "Cercenatorul3000" application was a crucial step in ensuring that all system components work harmoniously. This architecture includes the frontend, which handles the user interface and interaction flow, and the backend, which processes user requests and manages data. A robust database schema was designed to store project details, and data about regions and what cities, terrains and regions do those contain. Additionally, a web scraping module was incorporated to extract relevant information from the "Muntii Nostrii" website, and a recommendation engine was developed to provide personalized suggestions based on user preferences and historical data. The detailed architecture, outlining each component and their interactions, is discussed in section 3.3.

**3.2.3 Development Phase**

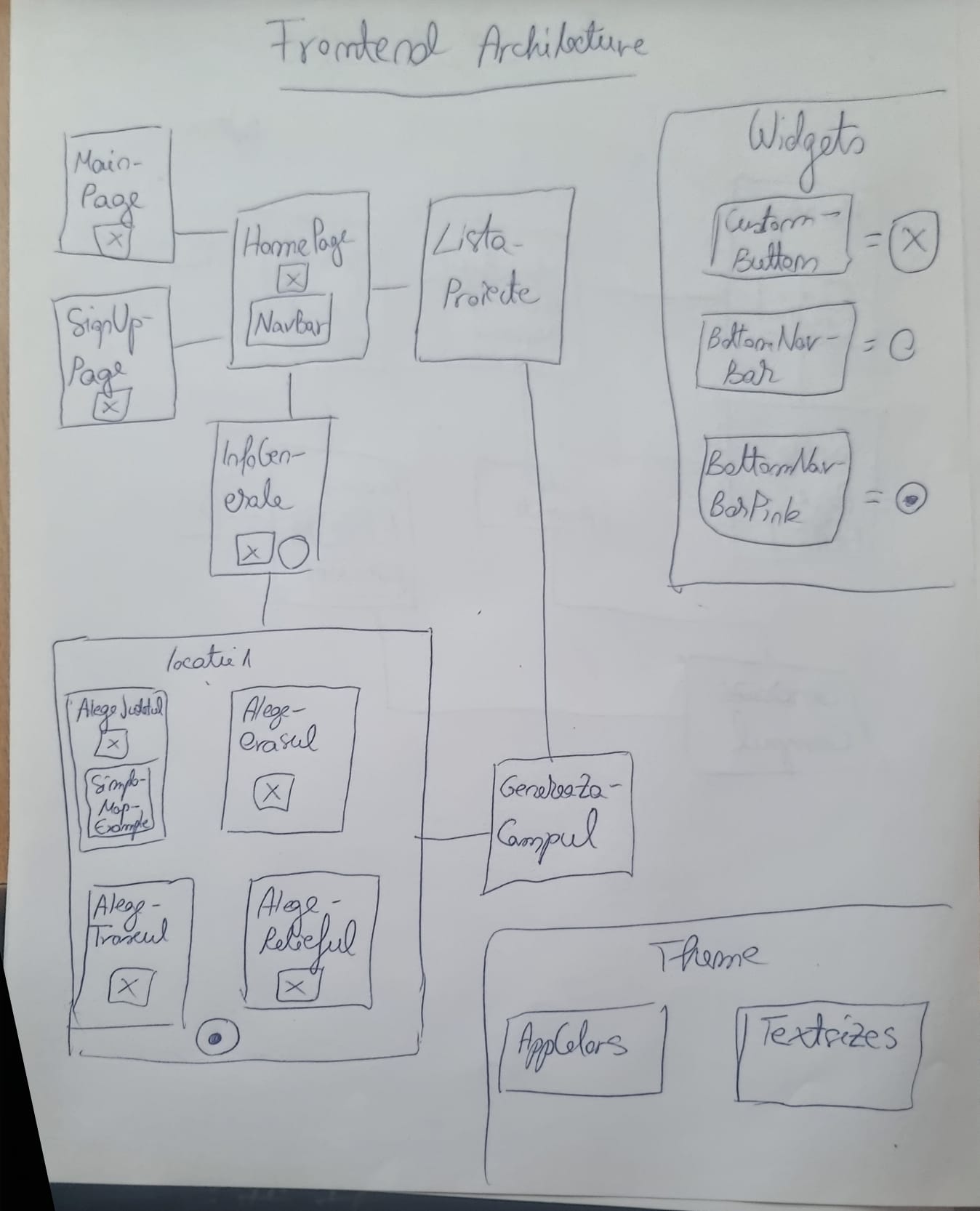
During the development phase, the focus shifted towards coding and implementing the various components outlined in the design phase. The process began with translating the UI/UX designs from Figma into Flutter, the framework chosen for the frontend development. This involved meticulously crafting user interfaces, ensuring they were visually appealing and intuitive for users to interact with. Simultaneously, efforts were directed towards building the web scraping module in Python to extract up-to-date trail information from "Muntii Nostrii" website. Following this, the recommendation engine was developed in Python, leveraging collaborative filtering algorithms to suggest personalized hikes and campsites based on user preferences. As the development progressed, attention turned towards setting up the database infrastructure. Firestore was selected as the database solution due to its scalability and seamless integration with Flutter. Finally, a Flask server was implemented to facilitate communication between the frontend and backend components. This allowed for seamless integration of the Python-based web scraping and recommendation engine with the Flutter frontend, enabling a cohesive and efficient application ecosystem. In section 3.3 there is a detailed overview of all the system components and in 3.4 their developement and implementation

**3.3 Overview of arhitecture**

In this section, we provide a detailed overview of the system architecture for the "Cercenatorul3000" application. The architecture is designed to ensure that all components work together harmoniously. This section will cover the architecture of the frontend, backend and database

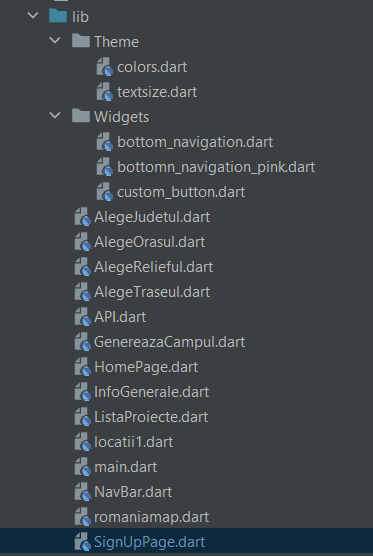
**3.3.1 Frontend Architecture**

The frontend architecture is centered around componentization, with each component representing an activity and a tree of widgets. The application is developed using Flutter within Android Studio.

****

1. Frontend Structure

To understand its architecture, let's look at the structure within the 'lib' directory, where we find three main sections: theme, widgets, and activities.



Theme:

This section contains two classes: one for color variables and another for text variables. These variables are utilized throughout all other components, ensuring consistency in the application's visual appearance.

Widgets:

Widgets are reusable parts of code present in one or more locations within the main activities. For example, the custom button widget is pre-designed to maintain consistency in the appearance of all application buttons. It is used in the majority of activities.

Activities:

The application comprises several components, with each component associated with an activity bound to a screen. We have a list of 13 activities which are the building blocks for the screens that create the frontend of the applications.

Further down is the way the screens bind to each other and how they can be navigated.

2. Navigation and Integration

The main screen, labeled “main”, is the first page you see upon entering the app housing the login functionality. From here, users have the option to navigate either to the sign-up page (SignUpPage) or directly to the homepage (“HomePage”). Similarly, upon accessing the sign-up page, users retain the flexibility to navigate back to either the login interface or go to the homepage.

Once on the homepage, users encounter a decision point where they can choose to explore general information (“InfoGenerale” page) or access a menu known as the navbar. This navbar allows users to navigate to the project list (“ListaProiecte”).

Opting to explore general information instead of entering the navbar signals the initiation of a new project.

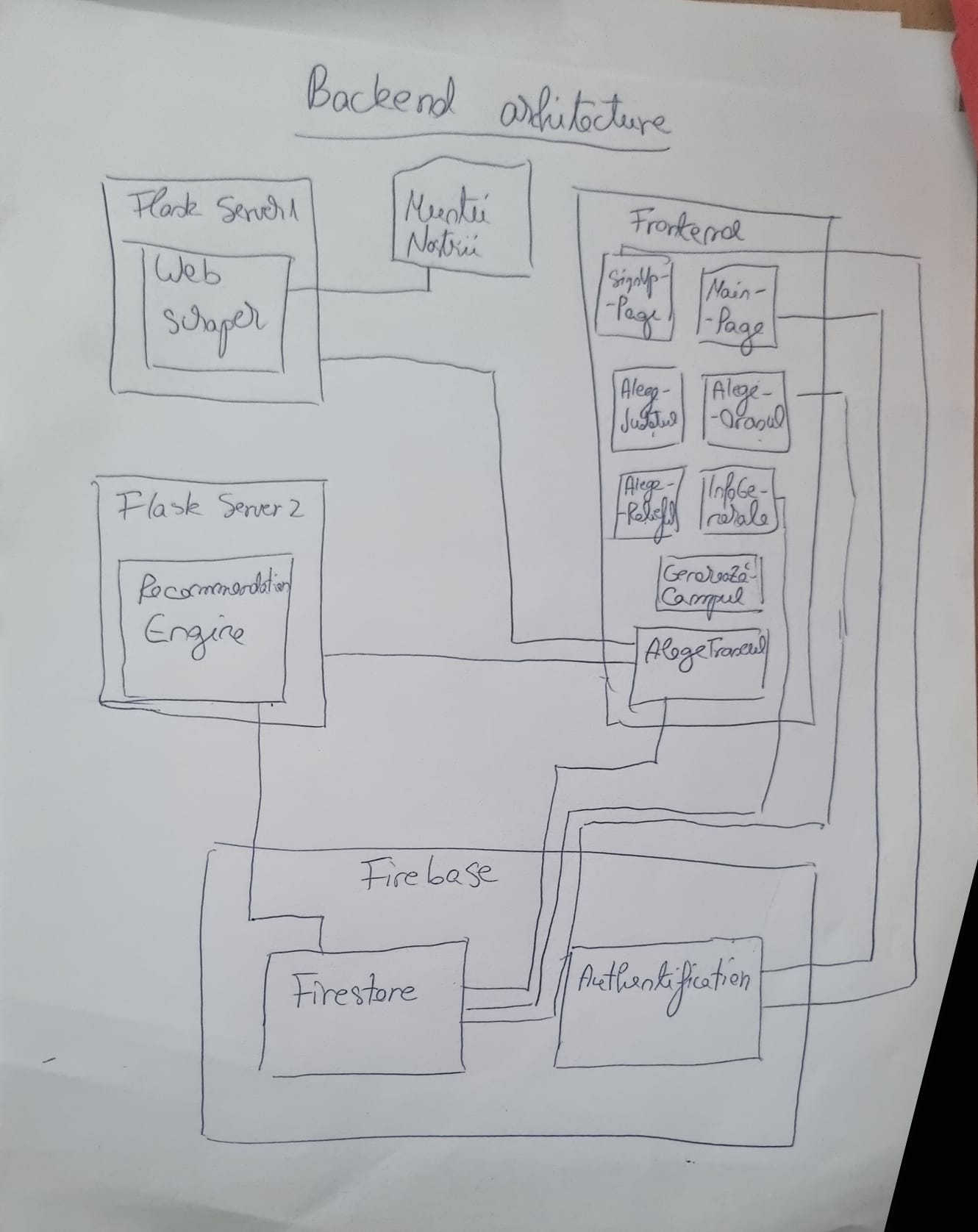
Users proceed to “locatii1”, which functions as another menu containing four distinct classes: choose county(“AlegeJudetul”), choose city(“AlegeOrasul), choose terrain(“AlegeRelieful”), and choose route(“AlegeTraseul”). Each class becomes visible based on arrow interactions, dynamically appearing on the main screen as users activate their selections.

Moreover, the choose county page includes the “romanianmap” class, utilized to design the page and generate a map for selecting counties. After navigating through one or more pages within the 'locations' menu, users can further advance to “GenereazaCampul”, accessible from both “ListaProiecte” and “locatii1”. Additionally “InfoGenerale” incorporates the “bottom\_navigation” widget from the widgets package, while “locatii1” itself contains the “bottom\_navigation\_pink” widget.

It's noteworthy that back buttons are strategically placed throughout the application, facilitating seamless navigation. From the “GenereazaCampul” page, users can navigate back to either “locatii1” or “ListaProiecte” (depending on their entry point). Similarly, from “locatii1”, users can return to “InfoGenerale” and from there to “HomePage”. Lastly, from the “ListaProiecte”, users can navigate back to the homepage navbar.

**3.3.2 Backend architecture**

The backend architecture of the hiking trails recommendation application is designed to efficiently handle data processing, deliver personalized experiences, and ensure seamless communication between various components. The key components of the backend include the Flask server, Firebase integration, the recommendation engine, and the web scraper. Our design ensures modularity and scalability, with each component focused on specific tasks while collaborating seamlessly within the larger system.

****

1. The Flask Server

In our code, the Flask server serves as the central component of the backend, handling URL routes and responding to HTTP requests received from the frontend and other services.

Our project utilizes two Flask servers, one to connect the frontend with the web scraper and another to connect the frontend with the recommendation engine. It's crucial that each port specified in the app.run() function is unique when using multiple Flask servers simultaneously on the same system. This is necessary to avoid port conflicts and to allow each server to listen on a specific port without interference.

1. Firebase Integration

Our project relies on Firebase for both database management and user authentication. This platform seamlessly integrates with our Flutter frontend, playing a crucial role in managing user authentication processes such as login and sign-up. Firebase seamlessly interacts with multiple components within our application, such as InfoGenerale, AlegeJudetul, AlegeOrasul, AlegeTraseul, and AlegeRelieful. It facilitates the transmission of data to and from these components. For instance, the former components send data to the database, while GenereazaCampul retrieves data from the database to populate fields or sections within the application. Furthermore, Firebase Firestore serves as a bridge to our recommendation engine, enabling the retrieval of project and hiking trail data from the database and the storage of personalized recommendations for users.

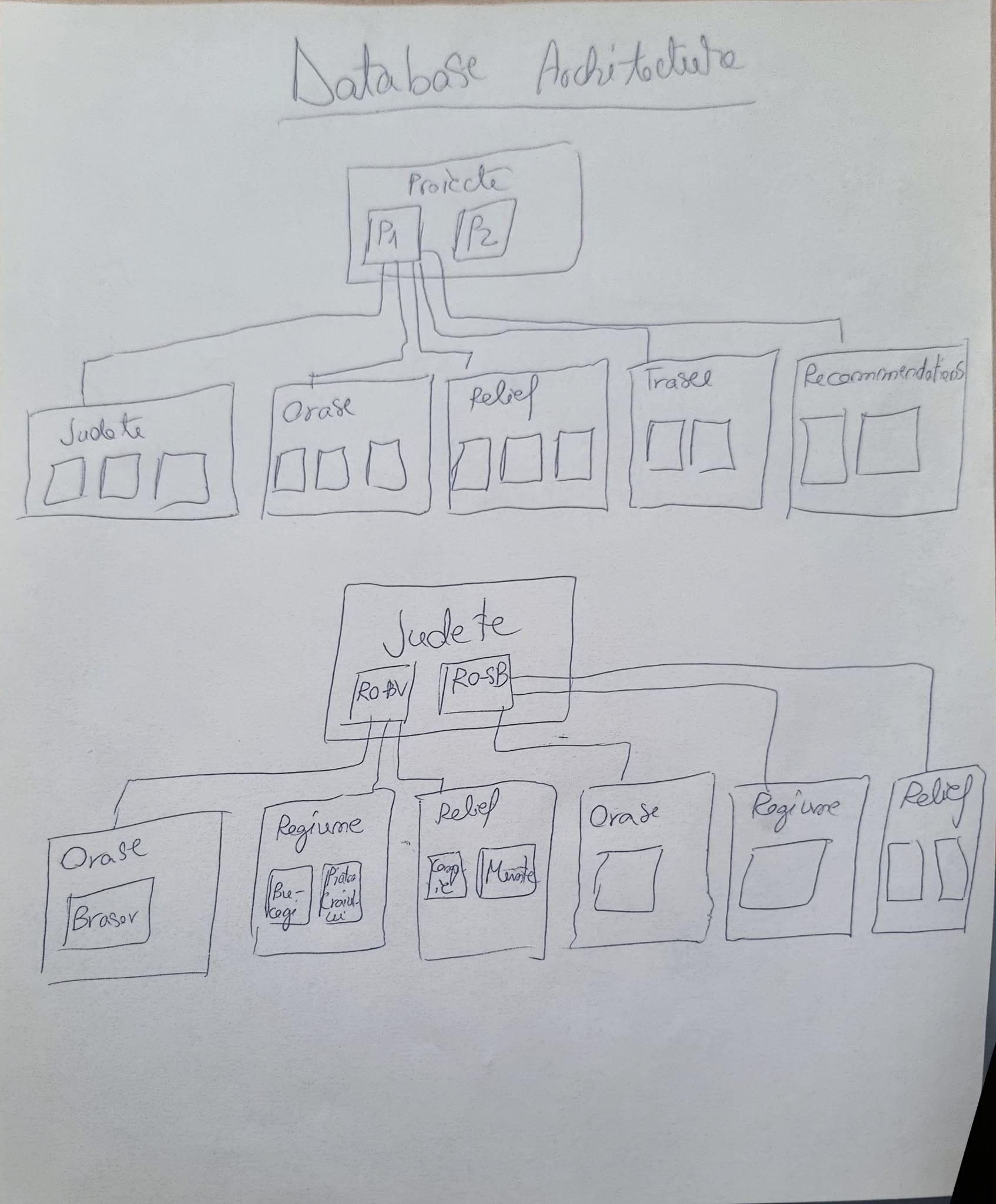
1. Recommendation Engine

The recommendation engine, implemented in Python, interfaces with one of the Flask servers within our architecture. This Flask server acts as a mediator between the frontend and the recommendation engine, handling HTTP requests and responses. The frontend communicates with the Flask server to trigger the recommendation process and retrieve the generated suggestions. Additionally, the recommendation engine accesses the Firebase Firestore database directly to fetch relevant data, such as user preferences and hiking trail details, necessary for generating recommendations. This architecture ensures a clear separation of concerns, with the recommendation logic encapsulated within the engine, while the Flask server manages the communication between the frontend and the backend components.

1. Web scraper

The web scraper component operates independently within our architecture, handling the extraction of data from external sources. Written in Python, the web scraper connects to a separate Flask server, distinct from the one linked to the recommendation engine. This Flask server acts as an intermediary between the frontend and the web scraper, facilitating the exchange of requests and responses. When triggered by user input or system events, the frontend communicates with this Flask server to initiate the scraping process and retrieve the scraped data. The Flask server then delegates these requests to the web scraper, which navigates through “Muntii Nostrii”, extracts relevant information, such as hiking trail details and formats it for further processing.

3.3.3 Database Arhitecture

****

Our database architecture primarily relies on Firestore Database. We have two main collections: "Judete" and "Proiecte." The "Judete" collection serves as a predefined repository where data is read but not added during the application's usage. Each document within this collection represents a specific county, and within each county, there are sub-collections for cities, terrain, and regions, each with predefined documents.

On the other hand, the "Proiecte" collection is dynamically populated based on user activity. When a user presses a button to create a new project, attributes such as name, number of people, and duration are stored. Each project can have anywhere from 0 to 5 sub-collections, depending on user input. These sub-collections include "Judete," "Orase," and "Relief," with each document representing a selected county, city, or terrain by the user.

Regarding hiking trails ("Trasee"), each trail is saved upon user interaction, utilizing data fetched from the web scraper. The document name corresponds to the trail's location, with attributes such as region, difficulty, and duration.

In the case of recommendations, the collection and documents are generated based on whether the user interacts with a button, dynamically tailoring recommendations accordingly.

**3.3.4 Data Flow**

1. User Interaction with the Frontend:

Users interact with the application interface developed in Flutter, triggering actions such as login, registration, or navigation between screens. The data from the frontend flows to the backend through the Flask server. The actions of pressing two buttons on the AlegeTraseul page generate distinct HTTP requests sent to two different Flask servers in the backend.

1. Backend Processing:

The first button is linked to the Flask server responsible for the recommendation engine. When pressed, this button sends a simple HTTP request to the recommendation engine server, which then processes and generates personalized hiking trail recommendations without needing additional data. The second button is connected to a different Flask server that handles the web scraping component. When this button is pressed, it sends an HTTP request containing detailed hike information such as duration, region, and difficulty to the web scraper server. This server then fetches relevant hiking trail data from external sources based on the user's specified criteria.

1. Backend Response to the Frontend:

After processing the user's requests the Flask servers send HTTP responses back to the frontend. The frontend receives these responses and updates the application.

1. Frontend interaction with the Database:

Data also flows between the frontend and the database. Data is extracted from the appropriate collections in Firebase and processed according to the user's requests. For example, when a user creates a new project, the project data is saved in the "Proiecte" collection.

1. Backend interaction with the Database:

The recommendation engine is also bind with the database. It accesses the Firestore to retrieve necessary user preferences and trail data, and then updates the database with the generated recommendations.

1. Integration of the Web Scraper:

When triggered by the frontend, the web scraper collects information about hiking trails and formats it for use in the application. The extracted data is then sent back to the Flask server, which processes it and forwards it to the frontend for display to the user.

**3.4 implementation**

**3.4.1 Frontend Implementation**

Our web application made in Flutter uses MaterialApp for design and MediaQuery for responsiveness. Each screen is composed of a UI part, which is structured as a widget tree and one or more functions that link the frontend with the database or the Flask server.

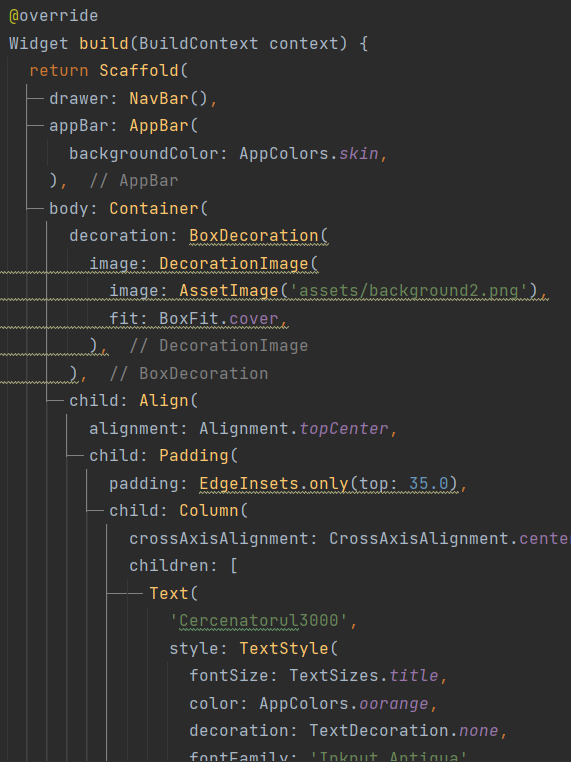


Figure 1 widget tree

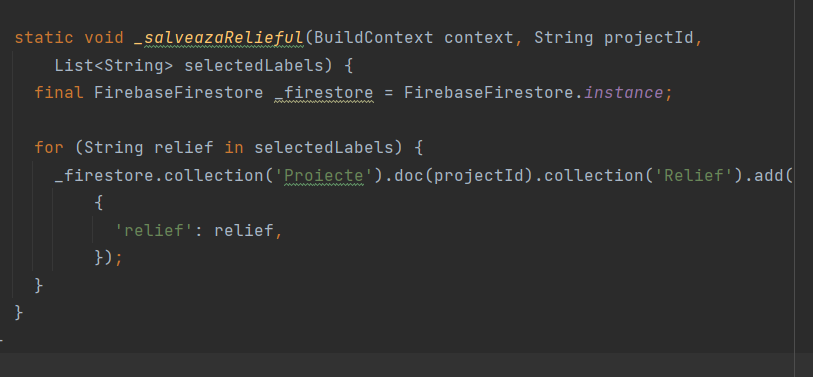


Figure 2 firesbase connection

Here we have a short description of each component of the app

The main.dart file serves as the entry point for the application and contains the initialization code for Firebase, as well as the definition of the MyApp and MainPage classes. The MyApp class is a StatelessWidget that configures the MaterialApp widget with the MainPage as the initial route and disables the debug banner. The MainPage class, on the other hand, is a StatefulWidget representing the login page of the application. It includes UI elements for entering the username and password, along with a login button. The user interaction is handled by the login method, which attempts to sign in using Firebase Authentication. If successful, the user is navigated to the HomePage. If an error occurs during login, a SnackBar widget displays the error message. Additionally, users have the option to navigate to the SignUpPage by tapping the 'Sign up!' text. The goToSignUpPage method handles this navigation action.

The SignUpPage allows users to register a new account with a username and password. The SignUpPage class is a StatefulWidget responsible for handling user interactions and managing state changes. It imports necessary packages such as 'package /material.dart' for Flutter widgets and 'package/firebase\_auth.dart' for Firebase authentication. The page layout includes a background image overlaid with a semi-transparent container for user input fields and buttons.TextFormField widgets capture user input for the username and password fields. Users can sign up by entering their desired username and password and pressing the 'Sign Up' button. The signUp() function attempts to create a new user account using Firebase Authentication. Success or failure is displayed via a SnackBar widget. Users can navigate back to the login page (MainPage) by tapping the 'Log in!' text or to the HomePage by tapping the Sign Up button.

The HomePage.dart file contains the implementation of the HomePage class, which represents the main interface of the application. It allows users to create new projects and provides navigation through a drawer menu. Within the HomePage class, the \_addNewProject method handles the creation of new projects. It interacts with Firestore to determine the number of existing projects, generates a unique project name, and adds the new project to the Firestore database. Upon successful creation, a SnackBar displays a success message, and the user is navigated to the InfoGenerale page to provide additional project details. The build method constructs the UI layout of the home page, including the app bar, background image, and UI elements for the application title, button, and navigation drawer. Users can tap the 'New Project' button to initiate the project creation process.

The NavBar represents a contextual menu that appears on the home screen (HomePage) of the application. This menu offers the user several options, including viewing saved projects and accessing a dialogue that explains how the application works. Within the build method, the contextual menu is constructed using a Drawer, which is a container that slides in from the edge of the screen and displays the available options. The UserAccountsDrawerHeader is used to display information about the user's account, such as the email address. If a user is authenticated, the email address is fetched from the current account used for authentication. The ListTiles represent each option in the menu. In the provided code, there are two ListTiles: The first ListTile has an icon and text indicating that the user can access saved projects. When this item is pressed, the application navigates to the ListaProiecte page, where saved projects are displayed. The second ListTile contains an icon and text indicating to the user that they can access an explanation of how the application works. When this item is pressed, a dialog titled 'How does it work?' is displayed, explaining the functioning of the application.

The ListaProiecte class orchestrates a screen in the application devoted to showcasing a catalog of projects sourced from a Firestore database. It orchestrates an AppBar housing the title "Lista Proiecte" (Project List) atop a layout primarily driven by a StreamBuilder. This dynamic widget is poised to react to changes in the Firestore collection named "Proiecte," ensuring real-time updates to the displayed project roster. Each project entry is encapsulated within a ListTile, presenting key details such as the project name, the number of associated persons, and the duration in days. The layout further integrates interactive features, empowering users to modify and delete individual projects via dedicated edit and delete buttons embedded within each tile. Supplementary functionality is provided through utility methods like \_showDeleteConfirmationDialog, facilitating seamless project management by enabling confirmation of deletion actions. Additionally, the \_updateProject method allows for the modification of project details with efficient updates to the Firestore database. Through these functionalities, the ListaProiecte screen enriches user engagement, streamlining project oversight and fostering a cohesive user experience within the application.

In the InfoGenerale screen, users input and update essential project details like name, number of persons, and duration, crucial for database management. Text form fields capture user input, with a save button triggering updates to the Firestore database. Upon success, a snackbar confirms the update, while a navigation button enables easy return to the previous screen. The bottom navigation bar has design purpose and is a reusable widget from the widgets directory.

In the locatii1 screen, the UI is composed of a Stack widget allowing overlapping elements. The main content area dynamically changes based on the \_text variable, which represents the current step in the location selection process. Depending on the step, different widgets like AlegeJudetul, AlegeOrasul, AlegeRelieful, and AlegeTraseul are displayed. These widgets are positioned within the Stack using the Positioned widget to ensure proper layout. Navigation between steps is facilitated by the row of buttons located at the top of the screen. The \_changeTextAndColor method updates the \_text variable and controls the navigation logic, allowing users to move forward or backward in the selection process. Additionally, the app bar contains navigation buttons to proceed to the next step or return to the previous screen. When users complete the location selection process, they can navigate to the GenereazaCampul screen by tapping the navigation button, passing along the projectId for further processing.

In the AlegeJudetul screen, users can select counties from a map. The selected counties are displayed dynamically at the top of the screen. The map itself is implemented using the SimpleMapExample widget from the romaniamap.dart file. When users make selections on the map, the list of selected counties is updated accordingly. Below the map, there is a "Save" button provided by the CustomButton widget, allowing users to save their selections. When this button is pressed, the \_salveazaJudetele method is called. This method saves the selected counties to the Firestore database under the specified project ID. Each selected county is stored as a document within a subcollection named "Judete" under the project document.

The SimpleMapExample widget displays a map of Romania with different regions highlighted in various colors. Users can interact with the map by tapping on different regions. When a region is tapped, its color changes to pink, indicating selection, and the onSelectionChanged callback is triggered, passing a list of the selected counties to the parent widget. The list of selected counties is constructed based on the regions with the pink color, and it is passed to the parent widget using the callback function. Internally, the map is implemented using the SimpleMap widget from the countries\_world\_map package, customized to display the map of Romania (SMapRomania)

The AlegeOrasul widget allows users to select cities from a list of available cities fetched from Firestore. Initially, the widget retrieves the list of cities associated with each county and populates the orase list. Users can then select or deselect cities by tapping on the corresponding buttons displayed using the Wrap widget. Selected cities are stored in the oraseSelectate list, and when the "Save" button is pressed, the \_salveazaOrasele function is called. This function iterates over the selected cities, adds them to the Firestore collection associated with the project, and displays a success message using a SnackBar. Additional actions can be added after saving the cities, such as navigation to another page or updating the widget's state.

The AlegeRelieful widget enables users to select relief types from a grid of available options fetched from Firestore. Initially, it retrieves the list of selected counties associated with the current project and collects the available relief types for those counties. Users can then tap on the relief buttons to toggle their selection. Selected relief types are stored in the selectedLabels list, and when the "Save" button is pressed, the \_salveazaRelieful function is called. This function iterates over the selected relief types and adds them to the Firestore collection associated with the project. A success message is displayed using a SnackBar upon saving the relief types.

The AlegeTraseul widget allows users to select parameters for a route, such as difficulty, duration, and region. These parameters are fetched from Firestore based on the selected counties associated with the project. Users can then tap on buttons to select the desired options. Once the parameters are selected, users can save the route by pressing the "Save" button. This triggers the saveDataToServer function, which sends a request to a Flask server running locally. The Flask server processes the request and returns recommendations for routes based on the selected parameters. Additionally, users can reset the selected parameters by pressing the "Alege alt traseu" button. This clears the selection and allows users to choose new parameters. Lastly, users can request recommendations by pressing the "Get Recommendations" button. This sends a request to the Flask server, which generates and returns recommendations based on predefined criteria.

The GenereazaCampul widget serves as a comprehensive display of project details sourced from Firestore. It meticulously organizes and presents essential project information, including the project name, the number of persons involved, the duration in days, associated counties (Judete), types of relief, recommendations, routes (Trasee), and cities. Utilizing asynchronous data fetching through FutureBuilder widgets, it ensures a responsive user interface while seamlessly retrieving and displaying project data.

**3.4.2 Backend Implementation**

1. Flask Implementation

For our Flask Server we first need to define the route we use to access it, then to handle the HTTP request we get and then generate a response. Here is the detailed implementation for our project:

Routes Definition: In our application, we define routes using the @app.route() decorator in the Flask application. For instance, @app.route('/api') defines a route for handling API requests related to hiking trails. These routes map incoming requests to specific functionalities or endpoints within the application.

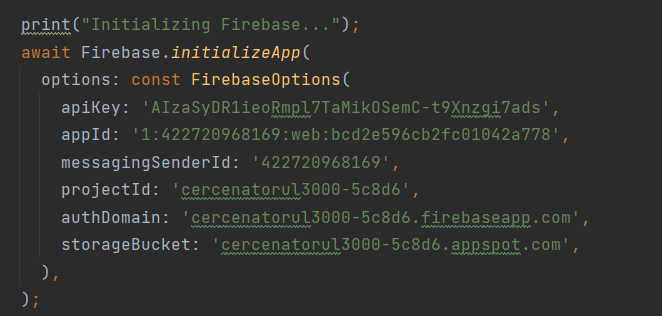
Request Handling: Within our Flask server, we utilize route functions to process incoming HTTP requests. For example, the hello\_world() function for the web scrapper handles GET requests to the /api endpoint. Inside this function, we extract data from the request's query parameters using request.args.get(). This data includes parameters such as duration, difficulty, and region, which are crucial for generating hiking trails. The /generate\_recommendations route, defined for both GET and POST methods, serves as the endpoint for generating recommendations. Upon receiving a POST request, it invokes the generate\_recommendations() function, which triggers the recommendation engine (collaborative\_filtering() in this case) to produce personalized recommendations.

Response Generation: After processing the request data, we generate a response using jsonify() to convert Python data structures into JSON format. This JSON response contains information about hiking trails based on the provided parameters. Each hiking trail's details, such as region, difficulty and hours, are included in the response to provide comprehensive results to the user.

Middleware Configuration: Our Flask server includes middleware components to enhance functionality and security. For instance, we configure CORS headers using response.headers.add(‘Access -Control-Allow-Origin’) to allow cross-origin requests, enabling our application to be accessed from different domains. Additionally, we may implement error handling middleware to catch and handle exceptions gracefully, ensuring a smooth user experience.

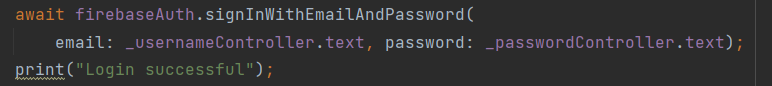
Firebase integration

Initialization: We initialize Firebase Firestore within our application using Firebase.initializeApp(). This step establishes a connection between our application and the Firestore database, enabling seamless data exchange.



Authentification: Within the MainPage widget, we handle the login process. When users attempt to log in, the login() function is invoked. This function utilizes firebase\_auth to authenticate users by their email and password.



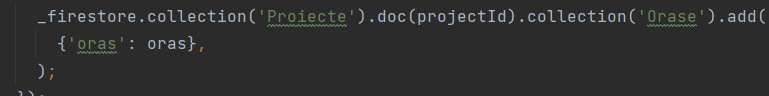


On the other hand, the SignUpPage widget handles the user registration process. The signUp() function is responsible for creating new user accounts using the createUserWithEmailAndPassword method provided by firebase\_auth.

Data Saving:

The first step involves creating a Firestore instance, obtained using \_firestore = FirebaseFirestore.instance. This provides access to Firestore functionalities within our Flutter application.

Next, the path where the data will be stored is defined. In this scenario, the data is saved in the "Proiecte" collection, under the document identified by projectId, and within the "Orase" subcollection. The add() method is used to insert a new document into the collection, with the city name stored as an attribute called "oras".



Data Retrieving:

To access Firestore functionalities within our Flutter application, we create a Firestore instance using FirebaseFirestore.instance. We then define the path for data retrieval: In this example, data is retrieved from the Firestore database under the "Proiecte" collection, within a specific document identified by projectId, and further within the "Judete" subcollection



We use the get() method to fetch the data from the specified path.

We access the document data using snapshot.data!.docs, which contains a list of QueryDocumentSnapshot objects representing each document retrieved from Firestore. For each document, we extract the value of the "judet" attribute using snapshot.data!.docs[index]['judet']. This retrieves the city name from each document.



1. Recommendation Engine

We start by calling the collaborative\_filtering() function, which initiates the collaborative filtering process for generating personalized recommendations. First it calls the the get\_projects\_and\_hikes() function. This function is called to fetch the list of projects and their associated hiking trails from the Firestore database. It initializes Firebase authentication and connects to Firestore. It retrieves project documents and for each project, obtains its associated trails, converts them into Hike instances, and adds them to the corresponding project's trails list. It returns the list of Project instances.

After that the get\_unique\_hikes(projects\_list) function is called: This function is called to obtain the list of unique hiking trails across all projects. It iterates through each project and each hike within the project, checking if the place attribute of the hike already exists in the list of unique hikes. If it doesn't exist, it adds the hike to the list of unique hikes. It returns the list of unique hiking trails.

After we have all the data from the database fetched and structured into a list of distinct projects and a list of distinct hikes we call the function initialize\_interaction\_matrix(projects\_list, unique\_hikes) Function:

This function is called to initialize the interaction matrix representing the relationship between projects and unique hiking trails. It creates a 2D numpy matrix where each row corresponds to a project and each column corresponds to a unique hiking trail. The value at position (i, j) is set to 1 if the i-th project includes the j-th hiking trail, otherwise 0. It returns the interaction matrix.

current\_project = projects\_list[-1]: This line selects the last project in the list of projects as the current project. current\_project\_index = len(projects\_list) - 1: It also stores the index of the current project in the list.

A dictionary project\_similarities is initialized to store the similarity scores between the current project and other projects. For each project in the projects\_list, excluding the current project: A list other\_project\_hikes is created containing the places of all hikes in the current project. common\_hikes is then populated with the places of hikes that are common between the current project and the other project. The similarity score is calculated as the ratio of the number of common hikes to the total number of hikes in the current project. This similarity score is stored in the project\_similarities dictionary with the project name as the key.

A dictionary recommendations is initialized to store recommendation scores for each unique hike. For each unique hike in the unique\_hikes list: If the current project hasn't interacted with the hike (as indicated by the interaction matrix): A list hike\_similarities is initialized to store similarity scores with other projects. For each project in the projects\_list: If the project has interacted with the hike (as indicated by the interaction matrix): The similarity score for the project (if available) is appended to hike\_similarities. The recommendation score for the hike is computed as the sum of similarity scores across all projects. This recommendation score is stored in the recommendations dictionary with the hike as the key.

sorted\_recommendations is created by sorting the recommendations dictionary based on recommendation scores in descending order.

Finally, the save\_recommendation\_to\_firestore() function is called to save the recommendations to the Firestore database under the current project's subcollection 'Recomandari'.

Finally, we save the generated recommendations to the Firestore database for future reference.

We create documents within the 'Recomandari' subcollection under the current project, storing recommended hiking trail details such as place, region, difficulty, kilometers, and duration.

Web scraper:

**3.3. cv Tree-Based Selection Process**

**The tree-based selection process is a method of organizing and making decisions that ensures a logical and efficient pathway by using a hierarchical structure similar to a tree. In this structure, each decision point (or node) branches out into multiple possible options (or child nodes), creating a clear and organized path for making complex decisions. Here’s a detailed explanation of how this process works:**

**\*Root Node\***

**The root node is the starting point of the tree. It represents the initial decision that needs to be made.**

**\*Branches and Child Nodes\***

**From the root node, the tree branches out into multiple child nodes, each representing a possible option for that decision point. Each child node then becomes a parent node to its own set of child nodes, representing further decisions that need to be made based on the initial choice.**

**\*Leaf Nodes\***

**The leaf nodes are the endpoints of the tree, where no further decisions are needed. These represent the final outcomes of the decision-making process.**

**Here's a detailed breakdown of how this works:**

**\*1. Choosing the County\***

**The first step in the tree is to choose the county where the camp will take place. This decision forms the root node of the tree.**

**\*Example:\***

**If the user selects Brasov County, this choice becomes the root node.**

**\*2. Choosing the Terrain or Region\***

**Once the county is selected, the next step is to choose the type of terrain or region. The terrain options are dynamically filtered based on the county chosen to ensure geographical accuracy.**

**\*Example:\***

**In the case of Brasov, available terrain types might include Bucegi and Piatra Craiului, excluding options like sea or delta because Brasov is not near these geographical features.**

**\*Tree Node Structure:\***

**- \*Root Node:\* County (e.g., Brasov)**

**- \*Child Nodes:\* Bucegi, Piatra Craiului (filtered based on county)**

**\*3. Determining the Hike and its Region\***

**The third step is to determine the hike and its region within the selected terrain. The application will present various hike options that are feasible and safe for camping activities, again filtered by the chosen county.**

**\*Example:\***

**If Piatra Craiului is selected as the terrain in Brasov, the application might suggest hikes through specific trails within that region.**

**\*Tree Node Structure:\***

**- \*Root Node:\* County (e.g., Brasov)**

**- \*Child Node:\* Terrain (e.g., Piatra Craiului)**

**- \*Child Nodes:\* Specific hikes or trails in Piatra Craiului**

PUNE POZA CU UN GRAF

3.5 Example Workflow