\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Programmer Documentation**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Introduction**

This project implements a robust search engine system, leveraging the power of Google Colab for development and Firebase as the backend for real-time data operations. Designed with modularity and scalability in mind, the system supports the following features:

**Manage Index:** Provides an interface for modifying, and deleting indexed terms and their associated URLs. This feature ensures the system remains relevant and up-to-date as the data evolves.

**Search Functionality:** Offers users the ability to execute queries with results ranked based on relevance. An integrated autocomplete feature enhances the search experience by suggesting terms as users type.

**Statistics Visualization:** Displays insights into search term trends using bar charts. This helps administrators or analysts identify popular queries and optimize the indexed content accordingly.

**Chatbot:** A conversational interface allows users to interact with the system in a human-friendly manner. The chatbot utilizes fuzzy matching to handle approximate inputs and typos, ensuring meaningful responses even with imperfect queries.

**Purpose of the System**

The goal of this project is to provide a comprehensive and interactive search platform for the site Huawei.com that combines data management, querying, visualization, and conversational interaction. This serves multiple use cases, including:

**Search Efficiency:** By preprocessing queries (e.g., stemming, stopword removal), the system ensures relevant results are returned efficiently.

**Data Insights:** Administrators can monitor search trends and optimize indexed data based on user behavior.

**User Engagement:** The chatbot interface enhances user interaction by providing answers to questions of the users. The chat bot works like hard codded.

**System Architecture**

The architecture of the system emphasizes a Model-View-Controller (**MVC**) design pattern:

**Model:** Firebase acts as the centralized database for storing terms, URLs, and search history.

**View:** The HTML/JavaScript frontend renders an intuitive user interface, ensuring seamless navigation across features.

**Controller:** Python functions, registered as callbacks in Google Colab, act as intermediaries, linking the frontend with the backend.

**Technological Highlights**

**Firebase:**

Provides real-time data management.

Handles CRUD (Create, Read, Update, Delete) operations for terms, URLs, and search history.

**Where It’s Used:**

**Manage Index Page:**

CRUD operations for managing terms and their URLs.

**Search Page:**

Reading term data for query processing.

**Statistics Page:**

Fetching top-searched terms for data visualization.

**Chatbot Page:**

Retrieving predefined responses for user queries.

**Google Colab:**

Colab supports the integration of Python with frontend components (HTML/JavaScript) via custom callbacks.

Python functions registered as callbacks can be invoked directly from JavaScript, bridging the gap between the frontend and backend.

**JavaScript,CSS and HTML:**

Power the dynamic frontend, ensuring smooth interaction with backend services.

HTML structures the content and layout of the pages.

CSS styles the interface, ensuring a clean and user-friendly design.

**JavaScript:**

Handles user interactions (e.g., button clicks, dropdown selections).

Sends requests to backend Python callbacks via Colab.

Dynamically updates content, such as tables and charts, without reloading the page.

**Where It’s Used:**

**Frontend Pages:**

**Main Page:** Navigation and UI rendering.

**Manage Index Page:** Dynamically updating term tables.

**Search Page:** Autocomplete and result display.

**Statistics Page:** Rendering charts.

**Chatbot Page:** Sending user queries and displaying responses.

**Natural Language Processing (NLP):**

Implements stemming and stopword removal to process queries effectively.

**How It Works:**

**Stemming:**

Words are reduced to their root forms using the Porter Stemmer.

Example: "running" → "run"; "eaten" → "eat".

**Stopword Removal:**

Common words like "the" and "is" are filtered out based on a predefined list, leaving only meaningful terms.

**Where It’s Used:**

**Search Page:**

During query preprocessing in SearchService.preprocess\_query().

**Fuzzy Matching:**

Fuzzy matching is a technique to find approximate matches between strings, useful for handling typos or variations in user input.

The system uses the FuzzyWuzzy library to calculate a similarity score between the user’s input and predefined terms.

Example:

Input: "computng"

Stored Term: "computing"

Similarity Score: 90% (above the threshold, e.g., 80%, so it matches).

**Where It’s Used:**

**Chatbot Page**:

Matching user queries to predefined terms in ChatService.chatbot\_response().

**Chart.js:**

Chart.js processes data in JSON format and renders it into dynamic charts (e.g., bar charts).

**Where It’s Used:**

**Statistics Page:**

Visualizing the most-searched terms and their frequencies.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**System Design and Page Breakdown**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1. Main Page**

The Main Page serves as the navigation hub for all system features. It includes four buttons:

**Manage Index:** To modify the indexed terms and URLs.

**Search:** To query the index dynamically.

**Statistics:** To visualize search term data.

**Chatbot:** To interact with the system in a conversational manner.

**Process Flow:**

**User Interaction:**

Clicking a button triggers the showPage(pageId) function.

**Frontend Handling:**

The function toggles visibility by adding/removing the hidden CSS class for respective sections.

**Result:**

Users are seamlessly redirected to the requested functionality.

**Key Benefits:**

**Clarity:** Simplifies navigation for users.

**Efficiency:** Minimizes distractions by showing only relevant sections.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**2. Manage Index Page**

This page allows users to maintain the integrity of indexed terms and their associated URLs stored in Firebase.

**Core Features and Processes**

**1. Adding a URL to a Term**

**Objective:** Enhance the index by associating new URLs with terms.

**Step-by-Step Process:**

**User Action:**

The user inputs a term and URL, then clicks "Add URL."

**Frontend Processing:**

The JavaScript function add\_url\_to\_term(event) sends the term and URL to the backend via the notebook.add\_url\_to\_term callback.

**Backend Handling:**

The callback calls TermService.add\_term(term, url), which interacts with FirebaseService.add\_url\_to\_term(term, url).

**Firebase Operations:**

Checks if the term exists using a GET request.

If it exists:

Adds the URL or increments its frequency.

If it doesn’t exist:

Creates a new term entry with the URL.

Sends a success or failure response.

**Frontend Updates:**

If successful:

The term's URL table updates dynamically.

A success message is displayed.

If failed:

An error message informs the user.

**2. Removing a Term**

**Objective:** Delete outdated or irrelevant terms and associated URLs.

**Step-by-Step Process:**

**User Action:**

The user selects a term from the dropdown and clicks "Remove Term."

**Frontend Processing:**

The remove\_Term(event) function sends a request to notebook.remove\_term.

**Backend Handling:**

The callback invokes TermService.remove\_term(term), which uses FirebaseService.remove\_term(term) to send a DELETE request to Firebase.

**Frontend Updates:**

Refreshes the dropdown list dynamically.

**Error Handling:**

If Firebase fails to delete the term, an error message is displayed, ensuring transparency.

**3. Viewing Term Details**

**Objective:** Provide insight into all URLs linked to a specific term.

**Process Flow:**

**User Action:**

The user selects a term from the dropdown.

**Frontend Processing:**

JavaScript calls loadTermDetails(term).

**Backend Handling:**

FirebaseService.find\_term\_data(term) retrieves URLs and their frequencies.

**Frontend Updates:**

Displays the data in a dynamic table for easy review.

**Relevance:**

Allows users to monitor and validate the integrity of the indexed data.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3. Search Page**

The Search Page provides dynamic query handling with autocomplete and ranked results.

**Core Features and Processes:**

**1. Query Preprocessing**

**Objective:** Clean and standardize user input for optimal search relevance.

**Steps:**

**Stopword Removal:**

Words like "the," "is," and "and" are filtered out using a predefined list, as they add little value to search accuracy.

**Stemming:**

Words are reduced to their root forms using the Porter Stemmer. For instance:

"Running," "runner," and "ran" → "run."

This ensures that searches match variations of a word.

**2. Autocomplete Suggestions**

**Objective:** Provide term predictions as the user types.

**Process Flow:**

**User Interaction:**

As the user types, the fetchAutocompleteSuggestions(query) function sends the query to notebook.get\_autocomplete\_suggestions.

**Backend Processing:**

The callback queries Firebase for terms matching the prefix.

Returns the results to the frontend.

**Frontend Updates:**

Displays the suggestions dynamically below the search box.

**3. Executing a Search**

**Objective:** Retrieve and display relevant results, ranked by frequency.

**Process Flow:**

The user enters a query and clicks "Search."

JavaScript triggers the performSearch(event) function, sending the query to notebook.search\_word.

**Backend Processing:**

The query undergoes preprocessing (stopword removal and stemming).

Firebase retrieves results for the processed terms.

Results are ranked by URL frequency.

Search history is updated in Firebase.

**Frontend Updates:**

Displays the ranked results as clickable links.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**4. Statistics Page**

The Statistics Page offers a visual representation of search trends.

**Core Features and Processes:**

**1. Fetching Data**

**Objective:** Retrieve the most-searched terms from Firebase.

**Process Flow:**

JavaScript calls loadTopSearchTerms(), sending a request to notebook.get\_top\_search\_terms.

The callback uses SearchService.get\_top\_search\_terms(limit) to fetch data from Firebase.

The results are returned to the frontend.

**2. Visualization**

**Objective:** Display data trends in an intuitive format.

**Process:**

**Frontend:**

A bar chart is rendered using Chart.js, showing the most-searched terms and their counts.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5. Chatbot Page**

The **Chatbot Page** provides a conversational interface, allowing users to interact with the system by typing queries in natural language. The chatbot leverages predefined responses stored in Firebase and implements fuzzy matching to handle approximate queries or typos. Additionally, the system is capable of processing multi-term queries and returning multiple relevant answers.

**Core Features and Processes**

**1. Fuzzy Matching**

**Objective:** To handle approximate queries and provide meaningful responses even when the user makes typographical errors.

**How It Works:**

The **FuzzyWuzzy** library calculates similarity scores between the user's input and predefined terms stored in Firebase.

Terms with a similarity score exceeding a certain threshold (e.g., 80%) are considered matches.

**2.Multi-Term Query Handling**

**Objective:** To extract and respond to multiple terms from a single input, ensuring comprehensive coverage of user queries.

**Process:**

The user's input is split into individual components.

Each component is processed separately using fuzzy matching to identify relevant terms.

If multiple terms match, the corresponding responses are aggregated and returned to the user.

**3. Generating Responses**

**Objective:** To provide accurate answers or default fallback responses when no matches are found.

**Process flow:**

**User Interaction:**

The user types a query and clicks "Send."

**Frontend Processing:**

The sendMessage(event) function sends the query to the backend via the notebook.chatbot\_response callback.

**Backend Processing:**

**Matching Terms:**

The ChatService.chatbot\_response(user\_message) function identifies relevant terms using fuzzy matching.

Multi-term queries are processed to extract all matching terms.

**Fetching Responses:**

Predefined responses for each matched term are retrieved from Firebase.

**Combining Responses:**

If multiple terms are matched, their responses are concatenated.

**Fallback Response:**

If no matches are found, a default response is returned, such as: "I'm sorry, I don't have an answer to that question."

**Frontend Updates:**

Displays the user query and chatbot response in the chat history.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_