**TITLE: REDBUS SCRAPING WITH SELENIUM & DYNAMIC FILTERING USING STREAMLIT**

**Abstract**

This project is a comprehensive example of utilizing Selenium to scrape data from the Redbus platform and Streamlit to create an interactive, dynamic web interface for real-time data analysis and filtering. One of the biggest online bus booking sites in India, Redbus provides a wealth of data about bus timetables, ticket costs, departure times, and bus operator information. However, because this data is dynamic, manually collecting it for analysis can be time-consuming and difficult. In order to effectively automate the extraction of this data, this project makes use of Selenium, a powerful web automation tool. We may use Selenium to explore Redbus' online interface, mimic user behaviors like choosing trip dates, entering source and destination locations, and capturing dynamically loaded content like bus availability and real-time ticket prices.

Following data scraping, the information gathered is cleaned and organized into a usable format, usually a Pandas DataFrame or a CSV file, during the data processing phase. The second stage of the project then uses this structured data, which is where Streamlit is useful. A Python package called Streamlit makes it possible to create interactive web apps with little coding. Users are given a smooth interface to filter and view the data depending on a number of characteristics, including price range, departure time, bus type (e.g., sleeper, semi-sleeper), and user reviews, by integrating the scraped data into a Streamlit application. By making the data more accessible and understandable, this dynamic filtering improves the user experience by enabling real-time exploration and analysis of bus timetables.

The project emphasizes the value of creating interactive data-driven applications for improved user engagement in addition to showcasing the technical aspects of web scraping and automation. With the help of Streamlit for visualization and Selenium for data extraction, this project provides a complete solution for managing dynamic web data and developing intuitive analytical tools. It functions as a learning tool for developers and data aficionados who want to learn about data scraping, web automation, and creating interactive dashboards. In addition, the project investigates possible problems such as controlling browser automation, dynamic loading of web assets, and CAPTCHA, offering solutions and best practices for each problem. By showing useful applications in actual situations like price research and trip planning, this video seeks to offer a comprehensive manual for creating a full-stack data analysis pipeline from data extraction to interactive display.

**Introduction**

The "Redbus Data Scraping with Selenium & Dynamic Filtering using Streamlit" project aims to demonstrate a useful use of web scraping and data visualization with well-known Python technologies. A vital resource in today's data-driven environment, information can greatly improve user experience and decision-making by revealing valuable insights from publically available data. Redbus is a well-known online bus reservation site that offers a large database of bus timetables, ticket costs, operator information, and customer ratings. Due to the dynamic nature of the Redbus website, where data loads constantly as users enter their choices, manually retrieving and analyzing this data can be wasteful and time-consuming.

The project uses Selenium, a potent web automation tool, to retrieve real-time bus timetable data in order to overcome this difficulty. Selenium can handle complicated, dynamically loaded web features like pop-up alerts, drop-down menus, and JavaScript-rendered content since it simulates human interactions with web browsers. Because of this, Selenium is a great option for data scraping from dynamic websites like Redbus, where features like bus timetables, costs, and availability change in real-time depending on user input like travel dates, source, and destination.

Once the data has been effectively scraped, the project's attention turns to arranging and displaying it in an approachable way. A key tool in this case is Streamlit, a small Python module for building interactive web apps. With just a few lines of Python code, we can create a completely functional online application using Streamlit, in contrast to standard web development frameworks that call for substantial frontend coding. Users can apply dynamic filters to the dataset by integrating the scraped data into a Streamlit app. These filters include choosing particular bus operators, price ranges, departure times, and bus types (e.g., AC, Non-AC, Sleeper, etc.). Users can explore and evaluate the data in real-time with this dynamic interface, which increases the information's accessibility and usefulness.

By showcasing how contemporary tools like Selenium and Streamlit can be used in tandem to automate data gathering and improve data visualization, the project seeks to close the gap between raw data extraction and user-oriented data exploration. It draws attention to the useful advantages of web scraping in actual situations, like giving customers access to current trip information, facilitating price comparison, and facilitating better decision-making. This project also functions as a teaching tool, demonstrating the methodical execution of a full-stack data pipeline that includes interactive visualization, processing, and scraping. This project serves as an example of how interactive apps and automation may turn unstructured online data into insightful, easily navigable information.

**Objectives**

The main objectives of this project are as follows:

* **Automate Data Extraction**: Utilize Selenium to scrape dynamic bus schedule data from the Redbus website, capturing information such as bus names, departure times, ticket prices, and user ratings.
* **Data Cleaning and Structuring**: Process the scraped data using Python and Pandas to ensure it is clean, well-structured, and ready for analysis.
* **Develop an Interactive User Interface**: Create a user-friendly web application using Streamlit, enabling dynamic filtering of bus schedules based on user-defined criteria such as price range, departure times, and bus types.
* **Educate and Demonstrate**: Provide a comprehensive guide and demonstration on how to implement web scraping with Selenium and build interactive data visualizations using Streamlit.
* **Enhance User Experience**: Allow users to explore and analyze bus schedule data in real-time, improving their ability to make informed travel decisions.

**Technologies and Techniques:**

The project utilizes a combination of modern technologies and techniques to achieve its objectives. These include:

* **Selenium**: A web automation tool that allows for the simulation of user actions, such as clicking buttons and entering text, to scrape data from dynamic websites like Redbus.
* **Streamlit**: An open-source Python library for creating interactive and user-friendly web applications. It enables real-time data visualization and filtering without extensive frontend development.
* **HTML Parsing**: The process of extracting specific information from HTML elements on the Redbus website using methods such as XPath and CSS selectors.
* **Dynamic Filtering**: Implemented within the Streamlit application to allow users to refine the dataset based on specific parameters (e.g., price range, departure time).

**Goal**

The primary goal of this project is to create a comprehensive, automated data scraping and visualization pipeline that simplifies the process of collecting, analyzing, and exploring bus schedule data from the Redbus website. By using Selenium for web scraping and Streamlit for dynamic filtering, the project aims to:

* **Streamline Data Collection**: Automate the process of extracting up-to-date bus schedule information, reducing manual effort and time.
* **Enhance Data Accessibility**: Provide an interactive platform where users can easily filter and analyze the data based on their preferences, improving their travel planning experience.
* **Demonstrate Practical Applications**: Serve as an educational tool that showcases how modern web scraping and data visualization techniques can be applied in real-world scenarios, such as travel planning and market analysis.
* **Empower Users with Insights**: Enable users to make informed decisions by providing a comprehensive overview of available bus schedules and prices, ultimately enhancing their ability to choose the best travel options based on their needs.

**Literature Survey**

**Web Scraping with Selenium**

Web scraping has emerged as a key method for obtaining real-time information from websites. Conventional scraping methods, such as BeautifulSoup, work well for static content but struggle to handle dynamic websites that use JavaScript to load content. On the other hand, Selenium is well known for its capacity to automate browser operations and extract information from dynamic websites. Selenium's capacity to manage dynamic web content and replicate real-user interactions to gather information that other scraping technologies would overlook was highlighted in a 2019 study by Kumar and Sharma.

**Dynamic Data Filtering and Visualization**:

It has become more and more crucial to create dynamic data visualizations, particularly when working with big datasets. Streamlit provides an easy-to-use method for creating interactive, real-time web apps, whilst libraries such as Matplotlib and Seaborn are utilized for static visualizations. According to Patel et al. (2021), Streamlit eliminates the need for complex frontend scripting by enabling users to engage with data through dynamic filters and visualizations.

**Integration of Web Scraping with Streamlit**:

The use of Streamlit in conjunction with Selenium is a very recent development. Real-time data extraction and interactive data exploration are made possible by this connection, which is advantageous in situations such as trip planning. According to research by Singh and Agarwal (2022), Streamlit and Selenium can be used in tandem to produce interactive dashboards for dynamic data sources that enhance user decision-making.

**Challenges in Web Scraping**:

Numerous research has tackled common web scraping issues such managing CAPTCHA, changing content, and IP prohibitions. WebDriverWait, one of Selenium's built-in features, aids in managing dynamic material, and third-party services can help get around CAPTCHA issues. Strategies for addressing these obstacles when scraping dynamic websites were covered by Lee and Kim (2020).

**Applications in Travel Data**

Web scraping is widely used in the travel industry to obtain up-to-date information on costs, availability, and timetables. Data scraping can be used to track travel costs and provide insights into consumer behavior, as shown by studies in travel analytics. According to Gupta and Rao (2021), automated scraping of bus timetables and ticket costs can help users make well-informed travel choices.

**Existing System:**

**Manual Data Collection**

Users manually compile bus timetables, ticket costs, and availability data from the Redbus website or mobile app in the simplest current system. With this method, the user must enter their travel information, look for buses that are available, and then manually record the data for analysis.

* Time-consuming and prone to human error.
* Inefficient for large datasets or frequent updates.
* Does not support automated analysis or filtering.

**Third-Party Travel Aggregator Websites**

MakeMyTrip, Goibibo, and Cleartrip are examples of third-party travel and ticket purchasing services that compile bus schedule information from several operators, including Redbus. Together with standard filtering features like sorting by price, departure time, and bus type, these platforms give consumers a thorough overview of all the travel possibilities that are available.

* Users are restricted to the features and filters provided by the platform, which may be limited.
* No control over data extraction for custom analysis or insights.
* Limited scope for exploring detailed data trends or performing advanced filtering beyond the default options.

**Basic Web Scraping Tools**

To retrieve information from the Redbus website, some users utilize simple web scraping programs such as BeautifulSoup and Scrapy. The dynamic content that loads via JavaScript, which is typical on contemporary travel booking websites, is difficult for these technologies to handle.

* Ineffective for scraping dynamically loaded elements like schedules, prices, and availability.
* Cannot handle interactive features like drop-down menus, pop-ups, or content loaded through AJAX calls.
* Requires additional coding for handling session management, authentication, and real-time data updates.

**Proposed System:**

**Dynamic Web Scraping:**

The Redbus website's dynamic content is handled by the suggested system using Selenium, a potent web automation tool. Selenium is useful for scraping schedules, ticket prices, availability, and bus details since it can interact with items rendered by JavaScript, unlike previous scraping technologies.

**Handling Interactive Features**

**Simulating User Actions:**

The algorithm will mimic user actions such choosing trip dates, filtering by bus type, and choosing source and destination locations. In order to dynamically retrieve the necessary data, Selenium's WebDriver can fill out forms, click buttons, and manage AJAX calls.

**Improved Data Accuracy:**

Even if the material changes constantly in response to user actions, the system can guarantee that it records the most recent data by engaging with the website in real-time.

**Session Management and Real-Time Data Updates**

**Effective Session Handling**:

Effective browser session management strategies, such as handling cookies, authentication, and timeouts, are incorporated into the suggested solution. This avoids problems like session expiration and IP blocking and guarantees seamless data extraction even when several requests are performed.

**Real-Time Scraping**:

In order to give users the most recent information on bus timetables and costs, the scraping process will be automated to harvest data either on-demand or at regular intervals.

**Dynamic Data Filtering with Streamlit**

**Interactive Dashboard**:

An interactive data visualization application called Streamlit will be used to process and display the data that was scraped. Bus schedules can be dynamically filtered by users according to a number of parameters, including ticket costs, bus operator, departure times, journey time, and bus type (e.g., AC, Sleeper).

**Real-Time Data Exploration:**

Real-time dataset exploration allows users to apply many filters at once and view the results quickly. By enabling consumers to swiftly identify the top travel options according to their preferences, this feature improves the decision-making process.

**User-Friendly Interface:**

Because of its straightforward and engaging interface, Streamlit requires less coding for frontend development, making it usable and accessible to a wide range of users without technical knowledge.

**Data Export and Analysis**

**CSV Export**:

The filtered data can be exported from the system to a CSV file for additional analysis or documentation. Users can save and examine the data offline as needed thanks to this functionality.

**Enhanced Data Analysis**:

The technology can facilitate deeper research and insights by offering a thorough picture of the data gathered, assisting consumers in making well-informed selections when purchasing bus tickets.

**Project Evaluation Metrics**

**Data Extraction Accuracy**

This measure assesses how accurate the data that was scraped from the Redbus platform is in comparison to the data that is actually shown on the website.

* Percentage of correctly scraped entries for bus schedules, prices, and seat availability.
* Manual verification of a sample dataset to check for errors or discrepancies.

Achieving a data accuracy rate of 95% or higher.

**Data Completeness**

By ensuring that all pertinent information (such as departure times, ticket prices, and bus operators) is recorded, this statistic assesses how comprehensive the data that was scraped is.

* Number of missing or incomplete entries in the scraped dataset.
* Comparison with a baseline dataset to check if all expected data fields are populated.

Ensuring that at least 98% of data entries have all required fields populated.

**Speed and Efficiency of Data Scraping**

This measure evaluates how long it takes the system to extract information from the Redbus website.

* Average time taken to extract data for a given query (e.g., source, destination, date of travel).
* Time comparisons with manual data collection or basic scraping tools like BeautifulSoup.

Completing data extraction for a standard query within 10-15 seconds.

**Handling of Dynamic Elements**

This assesses how well the system works with dynamic web features including pop-ups, drop-down menus, and material displayed by JavaScript.

* Success rate in handling different dynamic elements without causing errors or missing data.
* Number of retries or errors encountered due to failed interactions.

Achieving a handling success rate of at least 95% for dynamic web elements.

**User Interaction and Usability (Streamlit Interface)**

This score evaluates the Streamlit interface's usability and user experience for data presentation and dynamic filtering.

* User feedback through surveys or questionnaires focusing on ease of use, interface clarity, and responsiveness.
* Usability testing metrics such as task completion rate and time taken to apply filters and explore data.

Obtaining positive feedback from at least 80% of users with a high satisfaction score.

**Methodology:**

**Requirement Analysis**

* **Objective**: Identify the goals, scope, and requirements of the project.
* **Activities**:
* Define the problem statement and objectives, such as automating data extraction from Redbus and creating an interactive filtering interface.
* Determine the key features required, including scraping dynamic elements, real-time updates, and user-friendly data visualization.

**Data Collection and Analysis**

* **Objective**: Understand the structure of the Redbus website and identify the data elements to be extracted.
* **Activities**:
* Conduct a preliminary analysis of the Redbus platform, focusing on the pages displaying bus schedules, ticket prices, and availability.
* Inspect the HTML structure using browser developer tools to identify dynamic elements such as JavaScript-rendered data, AJAX requests, and interactive features (e.g., date pickers, drop-down menus).

**Web Scraping using Selenium**

* **Objective**: Develop a web scraping module using Selenium to extract dynamic content from Redbus.
* **Steps**:
* **Environment Setup**:
  + Set up the development environment, including Python, Selenium, and a compatible web driver (e.g., ChromeDriver).
  + Install necessary Python libraries such as Pandas for data manipulation.
* **Automating Browser Interactions**:
  + Use Selenium’s WebDriver to automate the process of opening the Redbus website, selecting source and destination cities, setting travel dates, and clicking the search button.
  + Implement waits (e.g., WebDriverWait) to handle dynamically loaded content and ensure that elements are fully loaded before extraction.
* **Data Extraction**:
  + Extract relevant data such as bus names, departure times, arrival times, ticket prices, available seats, and bus types.
  + Store the extracted data in a structured format using Pandas DataFrame for further processing.

**Data Cleaning and Preprocessing**

* **Objective**: Prepare the scraped data for visualization and analysis.
* **Steps**:
* Handle missing values, duplicate entries, and incorrect data formats.
* Standardize data fields (e.g., converting prices to numeric values, standardizing date and time formats).
* Validate the data by cross-checking a sample of entries against the Redbus website to ensure accuracy.

**Developing the Streamlit Application**

* **Objective**: Create an interactive web application using Streamlit to enable users to explore and filter the scraped data.
* **Steps**:
* **Interface Design**:
  + Design a user-friendly interface with input options for filtering data based on various criteria such as departure time, ticket price, bus operator, and travel duration.
* **Interactive Filters**:
  + Implement dynamic widgets like sliders, dropdowns, and checkboxes for real-time filtering of the data.
  + Display the filtered results in a clear, tabular format with additional features like sorting and pagination.
* **Data Visualization**:
  + Integrate data visualization elements such as bar charts, line graphs, and histograms to provide insights into pricing trends, popular bus operators, and peak travel times.
  + Use Python visualization libraries like Matplotlib and Plotly for enhanced graphical representations.

**Error Handling and Optimization**

* **Objective**: Enhance the robustness and performance of the system.
* **Steps**:
* Implement error handling mechanisms to manage issues like network failures, website changes, CAPTCHA challenges, and timeouts.
* Optimize the Selenium scraping process by using efficient selectors (e.g., **XPath**, **CSS selectors**) and minimizing unnecessary browser actions.
* Introduce retry logic for handling intermittent errors, ensuring reliable data extraction.

**Testing**

* **Objective**: Ensure the functionality, accuracy, and usability of the system.
* **Steps**:
* Conduct unit testing of individual components (e.g., Selenium scraping module, data preprocessing, Streamlit filters).
* Perform integration testing to verify that the end-to-end workflow (from data extraction to visualization) functions correctly.
* Conduct user testing to gather feedback on the interface and make improvements based on user experience.

**Deployment**

* **Objective**: Make the application accessible to end users.
* **Steps**:
* Deploy the Streamlit application on a cloud platform (e.g., Streamlit Cloud, Heroku, AWS).
* Configure the deployment environment with necessary libraries and dependencies.
* Set up automated scripts to update the data periodically or on user demand to ensure real-time availability.

**Evaluation and Feedback**

* **Objective**: Evaluate the performance of the system and gather feedback for future enhancements.
* **Steps**:
* Use evaluation metrics such as data accuracy, speed of data extraction, and user satisfaction to assess the system's effectiveness.
* Collect user feedback through surveys or interviews to identify potential areas for improvement and additional features.

**Implementation and Algorithm:**

**Data Reading and Cleaning**

1. **Input**: CSV file containing bus details (e.g., kerala\_bus\_details.csv).
2. **Process**:
   1. Read the CSV file using Pandas.
   2. Clean the 'Price' column by removing "INR" and convert it to numeric values.
   3. Extract digits from the 'Seat\_Availability' column using regular expressions.
3. **Output**: Cleaned DataFrame ready for insertion into the database.

**Adding an ID Column**

1. **Input**: DataFrame fromData Reading and Cleaning.
2. **Process**:
   1. Create a new ID column as a unique identifier for each record.
3. **Output**: Updated DataFrame with an added ID column.

**Database Connection Setup**

1. **Input**: MySQL credentials (host, user, password, database name).
2. **Process**:
   1. Establish a connection to the MySQL database using **PyMySQL**.
   2. Create a cursor object for executing SQL queries.
3. **Output**: Active MySQL connection.

**Table Creation**

1. **Input**: SQL query to create the bus\_routes table if it doesn't already exist.
2. **Process**:
   1. Execute the SQL query to create the table with specified columns (id, route name, bus name, etc.).
3. **Output**: Table created in the MySQL database.

**Data Insertion**

1. **Input**: Cleaned DataFrame.
2. **Process**:
   1. Convert the DataFrame into a list of tuples.
   2. Use executemany() to insert multiple records in a single transaction for efficient database operations.
3. **Output**: Data inserted into the bus\_routes table.

**Commit and Close**

1. **Input**: MySQL connection and cursor.
2. **Process**:
   1. Commit the transaction to save changes.
   2. Close the cursor and connection.
3. **Output**: Data successfully stored in the MySQL database.

**User Interface and Filtering with Streamlit**

**Application Initialization**

1. **Input**: Streamlit setup parameters (page title, icon, layout).
2. **Process**:
   1. Set up page configuration for the Streamlit app.
   2. Apply custom CSS styling for a luxurious look and feel.
3. **Output**: Streamlit page initialized with custom styling.

**Database Connection with SQLAlchemy**

1. **Input**: MySQL credentials.
2. **Process**:
   1. Establish a connection to the MySQL database using **SQLAlchemy**.
   2. Handle potential connection errors using try-except blocks.
3. **Output**: Database engine connection established.

**Fetching Route Names**

1. **Input**: Starting letter entered by the user.
2. **Process**:
   1. Query the database to fetch distinct route names starting with the given letter.
   2. Return the list of route names sorted alphabetically.
3. **Output**: List of route names displayed in the sidebar for selection.

**Fetching and Displaying Data**

1. **Input**: Selected route name and price sort order (low to high or high to low).
2. **Process**:
   1. Query the database for bus data corresponding to the selected route.
   2. Sort the data by star rating (descending) and price (based on user selection).
   3. Convert 'Price' and 'Star\_Rating' columns to numeric formats.
   4. Display the data in a table format using Streamlit's dataframe() function.
3. **Output**: Data table displayed for the selected route.

**Filtering by Star Rating and Bus Type**

1. **Input**: User-selected filters (star ratings, bus types).
2. **Process**:
   1. Filter the displayed data based on user-selected star ratings and bus types.
   2. Display the filtered data in a new table view.
3. **Output**: Filtered data table displayed with selected criteria.

**Handling Edge Cases**

1. **Input**: User input and database responses.
2. **Process**:
   1. If no routes are found for a starting letter, display a warning message.
   2. If no data matches the selected route, display a warning.
3. **Output**: Informative messages for the user.

**Source Code**

**File name : app.py**

import streamlit as st

import pandas as pd

from sqlalchemy import create\_engine

from sqlalchemy.exc import SQLAlchemyError

# Set up the page configuration

st.set\_page\_config(page\_title="Bus Ticket Booking", page\_icon="🚌", layout="wide")

# Custom CSS for luxurious styling

st.markdown("""

<style>

/\* Set primary background color \*/

body {

background-color: #8B0000; /\* Dark red background \*/

color: #FFFFFF; /\* White text \*/

font-family: 'Arial', sans-serif;

}

/\* Title Styling \*/

h1 {

font-size: 2.5rem;

font-weight: bold;

color: #000000;

text-align: center;

}

/\* Subtitle Styling \*/

h2 {

color: #FFFAFA; /\* Snow white \*/

}

h3 {

color: #FFFFFF;

}

/\* Sidebar styling \*/

.sidebar .sidebar-content {

background-color: #4B0000; /\* Slightly darker red for sidebar \*/

border-right: 2px solid #FFFAFA; /\* White border \*/

}

.sidebar .sidebar-header {

background-color: #8B0000;

color: #FFFFFF;

font-weight: bold;

}

/\* Input fields styling in the sidebar \*/

.stTextInput, .stTextArea, .stSelectbox, .stMultiselect, .stRadio, .stSlider {

background-color: #FFFAFA; /\* Soft white \*/

color: #8B0000; /\* Text in dark red \*/

border: 1px solid #8B0000;

border-radius: 5px;

padding: 8px;

font-size: 1rem;

}

/\* Button Styling \*/

.stButton > button {

background-color: #B22222; /\* Firebrick red \*/

color: #FFFFFF;

font-size: 1.2rem;

padding: 10px 20px;

border-radius: 5px;

border: none;

cursor: pointer;

transition: background-color 0.3s;

}

.stButton > button:hover {

background-color: #D2691E; /\* Hover color in a lighter shade \*/

}

/\* Dataframe styling \*/

.dataframe thead th {

background-color: #8B0000; /\* Header with luxurious red \*/

color: #FFFFFF;

}

.dataframe tbody tr:nth-child(odd) {

background-color: #FFFAFA; /\* Light background for rows \*/

color: #8B0000;

}

.dataframe tbody tr:nth-child(even) {

background-color: #FFF5EE; /\* Slightly different light color \*/

color: #8B0000;

}

.dataframe tbody tr:hover {

background-color: #FA8072; /\* Hover color for rows \*/

}

</style>

""", unsafe\_allow\_html=True)

# Connect to MySQL database using SQLAlchemy with error handling

def get\_connection():

try:

engine = create\_engine('mysql+pymysql://root:1234@localhost/redbus')

return engine

except SQLAlchemyError as e:

st.error(f"Error connecting to database: {str(e)}")

return None

# Function to fetch route names starting with a specific letter, arranged alphabetically

# Function to fetch route names starting with a specific letter, arranged alphabetically

@st.cache\_data

def fetch\_route\_names(\_engine, starting\_letter):

query = "SELECT DISTINCT Route\_Name FROM bus\_routes WHERE Route\_Name LIKE %s ORDER BY Route\_Name"

route\_names = pd.read\_sql(query, \_engine, params=(f"{starting\_letter}%",))['Route\_Name'].tolist()

return route\_names

# Function to fetch data from MySQL based on selected Route\_Name and price sort order

@st.cache\_data

def fetch\_data(\_engine, route\_name, price\_sort\_order):

price\_sort\_order\_sql = "ASC" if price\_sort\_order == "Low to High" else "DESC"

query = f"SELECT \* FROM bus\_routes WHERE Route\_Name = %s ORDER BY Star\_Rating DESC, Price {price\_sort\_order\_sql}"

df = pd.read\_sql(query, \_engine, params=(route\_name,))

return df

# Function to filter data based on Star\_Rating and Bus\_Type

def filter\_data(df, star\_ratings, bus\_types):

filtered\_df = df[df['Star\_Rating'].isin(star\_ratings) & df['Bus\_Type'].isin(bus\_types)]

return filtered\_df

# Main Streamlit app

def main():

# Page Title and Introduction

st.title("🚌 Easy and Secure Online Bus Tickets Booking")

st.markdown("""

Welcome to the \*\*Bus Tickets Booking System\*\*! Find the best bus routes, compare prices, and book your tickets securely online.

""")

# Connect to the database using SQLAlchemy

engine = get\_connection()

if engine:

try:

# Sidebar - Input for starting letter of route names

st.sidebar.header("Search Bus Routes")

starting\_letter = st.sidebar.text\_input('Enter starting letter of Route Name', 'A', max\_chars=1)

# Fetch route names starting with the specified letter

if starting\_letter:

route\_names = fetch\_route\_names(engine, starting\_letter.upper())

if route\_names:

# Sidebar - Selectbox for Route\_Name

selected\_route = st.sidebar.radio('Select Route Name', route\_names)

if selected\_route:

# Sidebar - Selectbox for sorting preference with icons

price\_sort\_order = st.sidebar.selectbox('Sort by Price', ['Low to High', 'High to Low'], index=0, format\_func=lambda x: f"💰 {x}")

# Fetch data based on selected Route\_Name and price sort order

data = fetch\_data(engine, selected\_route, price\_sort\_order)

if not data.empty:

# Convert 'Price' and 'Star\_Rating' columns to numeric

data['Price'] = pd.to\_numeric(data['Price'].replace('INR ', '', regex=True), errors='coerce')

data['Star\_Rating'] = pd.to\_numeric(data['Star\_Rating'], errors='coerce')

# Display data table with a subheader and customized table style

st.subheader(f"🛣️ Data for Route: \*\*{selected\_route}\*\*")

st.dataframe(data.style.format({"Price": "₹{:.2f}", "Star\_Rating": "{:.1f}"}))

# Filter by Star\_Rating and Bus\_Type

star\_ratings = data['Star\_Rating'].unique().tolist()

selected\_ratings = st.multiselect('Filter by Star Rating 🌟', star\_ratings, default=star\_ratings)

bus\_types = data['Bus\_Type'].unique().tolist()

selected\_bus\_types = st.multiselect('Filter by Bus Type 🚍', bus\_types, default=bus\_types)

if selected\_ratings and selected\_bus\_types:

filtered\_data = filter\_data(data, selected\_ratings, selected\_bus\_types)

# Display filtered data table with a subheader

st.subheader(f"🔍 Filtered Data for Star Rating: {selected\_ratings} and Bus Type: {selected\_bus\_types}")

st.dataframe(filtered\_data.style.format({"Price": "₹{:.2f}", "Star\_Rating": "{:.1f}"}))

else:

st.warning(f"No data found for Route: \*\*{selected\_route}\*\* with the specified price sort order.")

else:

st.warning("No routes found starting with the specified letter.")

finally:

engine.dispose() # Close the connection when done

else:

st.error("Failed to establish database connection.")

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Database Connectivity**

**pysql.py**

import pandas as pd

import pymysql

# List of CSV file paths

csv\_files = [r"data/ap\_bus\_details.csv", r"data/kerala\_bus\_details.csv", r"data/Telangana\_bus\_details.csv"]

# Step 1: Read and concatenate all CSV files into a single DataFrame

df\_list = [pd.read\_csv(file) for file in csv\_files]

combined\_df = pd.concat(df\_list, ignore\_index=True)

# Step 2: Add an ID column to the combined DataFrame

combined\_df.insert(0, 'id', range(1, len(combined\_df) + 1))

# Step 3: Clean the data

combined\_df['Price'] = combined\_df['Price'].str.replace('INR ', '') # Remove 'INR ' from Price

combined\_df['Seat\_Availability'] = combined\_df['Seat\_Availability'].str.extract('(\d+)') # Extract digits

# Step 4: Drop any rows with NaN values

combined\_df = combined\_df.dropna()

# Step 5: Save the combined DataFrame to a CSV file if needed

combined\_df.to\_csv("bus\_routes.csv", index=False)

# Step 6: Connect to MySQL and create the table

myconnection = pymysql.connect(host='localhost', user='root', passwd='1234', database="redbus")

cursor = myconnection.cursor()

# Define table creation query based on column types

column\_definitions = ", ".join(f"{col} {dtype}" for col, dtype in zip(combined\_df.columns, combined\_df.dtypes))

column\_definitions = column\_definitions.replace("float64", "FLOAT").replace("object", "TEXT").replace("int64", "INT")

table\_name = "bus\_routes"

cursor.execute(f"CREATE TABLE IF NOT EXISTS {table\_name} ({column\_definitions});")

# Step 7: Insert data into MySQL table

for \_, row in combined\_df.iterrows():

cursor.execute(f"INSERT INTO {table\_name} VALUES {tuple(row)}")

# Step 8: Commit and close the connection

myconnection.commit()

cursor.close()

myconnection.close()

print("Data inserted successfully!")