## **Redbus Data Scraping with Selenium & Dynamic Filtering using Streamlit**

### **1. Introduction**

**Project Overview:**

This project involves scraping data from the Red bus website using Selenium, storing the scraped data in an MySQL database, and developing a Streamlit application to dynamically filter and display the data. The goal is to automate the data extraction process and provide a user-friendly interface for data exploration.

**Objectives:**

* To scrape bus route details from Redbus.
* To store the scraped data in an MySQL database.
* To create a Streamlit application for dynamic filtering and visualization of the data.

### **2. Tools and Technologies**

* **Python:** For scripting and data manipulation.
* **Selenium:** For web scraping.
* **MySQL:** For data storage.
* **Streamlit:** For creating the web application.
* **Jupyter Notebook:** For development and testing.
* **Visual studio :** For development and testing.(streamlit)

### **3. Project Setup**

**Prerequisites:**

* Python installed on your machine.
* Required Python libraries: selenium, streamlit, pandas, pymysql.

**Installation:**

pip install selenium pandas pymysql streamlit

### 

### **4. Web Scraping with Selenium**

The code performs web scraping of bus transport data from the Redbus website and stores the scraped data in a MySQL database. It utilizes the Selenium library to interact with the web pages, extract relevant information, and handle dynamic content. The pymysql library is used to connect to the MySQL database and store the scraped data.

### **Web Scraping Process**

1. Initialization:

Imported required libraries and The Chrome WebDriver is initiated and maximized window and opened the required red bus website page

driver = webdriver.Chrome()

driver.maximize\_window()

driver.get('https://www.redbus.in/online-booking/uttar-pradesh-state-road-transport-corporation-upsrtc/?utm\_source=rtchometile')

1. Created one empty list to store the all the bus route details

# List to store the collected bus route data

bus\_route\_data = []

1. Created one function (route\_details()) to collect the route details in first page:

Webdriverwait –ensures that the page has loaded and that elements with the class route\_link are present before proceeding. Here Extracted all route\_link and route name in that screen and stored and returned as a list of dictionaries. I have used Try and Exception block, if anything goes wrong exception will catch the error.

# Function to collect route details from the page

def route\_details():

try:

WebDriverWait(driver.until(EC.presence\_of\_element\_located((By.CLASS\_NAME, "route\_link")))

route\_elements = driver.find\_elements(By.CLASS\_NAME, "route")

route\_links = [route.get\_attribute("href") for route in route\_elements]

route\_names = [route.get\_attribute("title").strip() for route in route\_elements]

# Collect the route details for this page and return them

return [{'route\_name': name, 'route\_link': link} for name, link in zip(route\_names, route\_links)]

except Exception as e:

print(f"An error occurred: {e}")

return []

1. Scrape\_all\_page()

The scrape\_all\_pages() function collects route links from multiple pages of a website, assuming there are up to 5 pages. It initializes an empty list all\_routes to store the route details and iterates through the pages using a for loop. For each page, if the page number is greater than 1, it waits for the pagination button to be clickable, scrolls to it, and clicks to navigate to the next page. After ensuring the page is loaded, it calls the route\_details() function to extract the route names and links, adding them to the all\_routes list. The function also includes error handling to catch and report any issues during the scraping process without crashing. Finally, it returns the complete list of route details collected from all pages.

def scrape\_all\_pages():

all\_routes = []

try:

# Loop through the first 5 pages (or adjust the range as needed)

for page in range(1, 6): # Assuming there are 5 pages

if page > 1:

# Wait for the pagination tab of the current page to be clickable

pagination\_tab = WebDriverWait(driver, 10).until(

EC.presence\_of\_element\_located((By.XPATH, f"//div[contains(@class, 'DC\_117\_pageTabs')][text()='{page}']"))

)

# Scroll to the pagination tab and click it

driver.execute\_script("arguments[0].scrollIntoView();", pagination\_tab)

driver.execute\_script("arguments[0].click();", pagination\_tab)

time.sleep(5) # Wait for the new page to load

# Collect the route details after navigating to the page (either the first page or after pagination)

all\_routes.extend(route\_details())

except Exception as e:

print(f"Error occurred while accessing page {page}: {str(e)}")

return all\_routes

1. scrape\_bus\_details function:

The scrape\_bus\_details(route\_name, route\_link) function is designed to extract detailed bus information from a specific route page on a website. It begins by navigating to the provided route\_link, waits for the page to load, and attempts to click the "View Buses" button, if present. The function then scrolls down to ensure that all relevant content is loaded and visible. After that, it extracts various details about the buses on the route, such as bus names, bus types, departure times, durations, reaching times, star ratings, prices, and seat availability, by locating elements on the webpage using CSS selectors.

# Function to scrape bus details for each route

def scrape\_bus\_details(route\_name, route\_link):

try:

driver.get(route\_link)

time.sleep(5)

# Try to click the "View Buses" button if available

try:

view\_buses = WebDriverWait(driver, 10).until(

EC.element\_to\_be\_clickable((By.CLASS\_NAME, "button"))

)

driver.execute\_script("arguments[0].click();", view\_buses)

time.sleep(5)

except:

print(f"No 'View Buses' button found for {route\_link}")

driver.execute\_script("window.scrollTo(0, document.body.scrollHeight);")

time.sleep(5)

# Extract data from the page

bus\_names = driver.find\_elements(By.CSS\_SELECTOR, "div.travels.lh-24.f-bold.d-color")

bus\_types = driver.find\_elements(By.CSS\_SELECTOR, "div.bus-type.f-12.m-top-16.l-color.evBus")

departure\_times = driver.find\_elements(By.CSS\_SELECTOR, "div.dp-time.f-19.d-color.f-bold")

durations = driver.find\_elements(By.CSS\_SELECTOR, "div.dur.l-color.lh-24")

reaching\_times = driver.find\_elements(By.CSS\_SELECTOR, "div.bp-time.f-19.d-color.disp-Inline")

star\_ratings = driver.find\_elements(By.CSS\_SELECTOR, "div.lh-18.rating.rat-red")

prices = driver.find\_elements(By.CSS\_SELECTOR, "div.fare.d-block")

seat\_availabilities = driver.find\_elements(By.CSS\_SELECTOR, "div.seat-left.m-top-16")

Once the data is collected, the function stores each bus's information in a dictionary, with keys like "Route Name", "Bus Name", "Bus Type", and more, adding each dictionary to the bus\_route\_details list. If the required information (e.g., bus type or star rating) is missing for a particular bus, the function handles this by assigning a default value like "N/A". Finally, the function returns the list of bus details. If any error occurs during the process, such as an issue loading the page or clicking a button, it catches the exception, prints an error message, and returns an empty list, ensuring the program doesn't crash.

bus\_route\_details = []

# Loop through each bus and store its details

for i in range(len(bus\_names)):

bus\_detail = {

"Route Name": route\_name,

"Route Link": route\_link,

"Bus Name": bus\_names[i].text,

"Bus Type": bus\_types[i].text if i < len(bus\_types) else "N/A",

"Departure Time": departure\_times[i].text if i < len(departure\_times) else "N/A",

"Duration": durations[i].text if i < len(durations) else "N/A",

"Reaching Time": reaching\_times[i].text if i < len(reaching\_times) else "N/A",

"Rating": star\_ratings[i].text if i < len(star\_ratings) else "N/A",

"Price": prices[i].text if i < len(prices) else "N/A",

"Seats Available": seat\_availabilities[i].text if i < len(seat\_availabilities) else "N/A"

}

bus\_route\_details.append(bus\_detail)

return bus\_route\_details

except Exception as e:

print(f"Error occurred while accessing {route\_link}: {str(e)}")

return []

1. scrape all details in all pages

The code first calls the scrape\_all\_pages() function to gather all route details from multiple pages, storing the result in the routes list. Then, for each route in the routes list, it calls the scrape\_bus\_details() function to extract detailed information about buses available on that route. If bus details are successfully scraped, they are added to the bus\_route\_data list using the extend() method, which ensures that the list grows as new bus details are appended for each route. This process efficiently collects bus data across all available routes.

# Scrape all route details

routes = scrape\_all\_pages()

# For each route, scrape bus details and add to the bus\_route\_data list

for route in routes:

bus\_details = scrape\_bus\_details(route["route\_name"], route["route\_link"])

if bus\_details:

bus\_route\_data.extend(bus\_details) # Add bus details to the bus\_route\_data list

1. saved to CSV file:

converted bus details data to pandas dataframe and saved dataframe to csv file t

# Convert bus\_route\_data to a pandas DataFrame

df = pd.DataFrame(bus\_route\_data)

# Save the DataFrame to a CSV file

df.to\_csv('Utter\_pradesh\_bus\_details.csv', index=False)

# Optionally, print the collected bus details

print("Data saved to Utter\_pradesh\_bus\_details.csv")

# Close the driver after scraping

driver.quit()

**5. SQL Data Storage Process**

* The code first sets the folder\_path to the location of the CSV files using a raw string (r'...') to ensure backslashes in the Windows file path are treated literally.

# path details of all csv files

folder\_path = r'C:\Users\HP\Desktop\kani\project\_redbus\Red\_bus\_final\_details'

* It then uses the glob module to find all CSV files in that folder and reads each file into a list of DataFrames (dfs). After reading the CSV files, it concatenates them into a single DataFrame (merged\_df) and saves the merged data into a new CSV file (all\_state\_bus\_detail.csv).

# Use glob to find all csv files in the folder

csv\_files = glob.glob(os.path.join(folder\_path, "\*.csv"))

dfs=[]

# Read each csv file and print or store the content

for file in csv\_files:

df = pd.read\_csv(file)

dfs.append(df)

merged\_df = pd.concat(dfs, ignore\_index=True)

merged\_df.to\_csv("all\_state\_bus\_detail.csv", index=False)

* Data cleaning and data type changes as per the SQL data type format

first reads the CSV file into the DataFrame. Then, it processes the 'Price' column by removing the 'INR' text and converting the values to numeric, coercing any non-numeric values into NaN.

### data cleaning and data type changes:

df = pd.read\_csv("all\_state\_bus\_detail.csv")

df['Price'] = df['Price'].str.replace('INR', "", regex= False)

df['Price'] = pd.to\_numeric(df['Price'], errors = 'coerce')

* The 'Seats Available' column is renamed to 'Seat\_Availability' and the numeric seat availability values are extracted from strings using a regular expression. These values are then converted to numeric, ensuring any non-numeric entries are set as NaN. The 'Seat\_Availability' column is explicitly cast to integers.

df.rename(columns={'Seats Available': 'Seat\_Availability'}, inplace=True)

df['Seat\_Availability'] = df['Seat\_Availability'].str.extract(r'(\d+)')

df['Seat\_Availability'] = pd.to\_numeric(df['Seat\_Availability'], errors = 'coerce')

df['Seat\_Availability'] = df['Seat\_Availability'].astype(int)

* Several other columns, such as 'Route Name', 'Route Link', 'Bus Name', 'Bus Type', 'Departure Time', 'Duration', and 'Reaching Time', are converted to strings. Finally, any rows containing missing values (NaN) are dropped using the dropna() method, resulting in a cleaned dataset.

df['Route Name'] = df['Route Name'].astype(str)

df['Route Link'] = df['Route Link'].astype(str)

df['Bus Name'] = df['Bus Name'].astype(str)

df['Bus Type'] = df['Bus Type'].astype(str)

df['Departure Time'] = df['Departure Time'].astype(str)

df['Duration'] = df['Duration'].astype(str)

df['Reaching Time'] = df['Reaching Time'].astype(str)

df= df.dropna()

* Connecting to the MySQL database

The script establishes a connection to the MySQL database using pymysql.

# Connect to MySQL

try:

myconnection = pymysql.connect(

host='127.0.0.1',

user='root',

passwd='\*\*\*\*\*\*',

database='Red\_Bus'

)

cursor = myconnection.cursor()

print("Connection successful!")

except pymysql.OperationalError as e:

print(f"OperationalError: {e}")

This code is used to create a table named Bus\_routes in the MySQL database if it doesn't already exist. The table is defined with the following columns:

* ID: An auto-incrementing primary key of type INT.
* Route\_Name: A column to store the name of the bus route as text (TEXT).
* Route\_Link: A column to store the URL or link to the route as text (TEXT).
* Bus\_Name: A column to store the name of the bus as text (TEXT).
* Bus\_Type: A column to store the type of bus (e.g., AC, Non-AC) as text (TEXT).
* Departing\_Time: A column to store the bus's departure time as TIME.
* Duration: A column to store the duration of the journey as text (TEXT).
* Reaching\_Time: A column to store the bus's reaching time as TIME.
* Star\_Rating: A column to store the bus's star rating as a floating-point number (FLOAT).
* Price: A column to store the price of the bus ride as a decimal number with two decimal places (DECIMAL(10, 2)).
* Seat\_Availability: A column to store the number of available seats as an integer (INT).

After defining the table structure in the create\_table\_query, it is executed using cursor.execute(create\_table\_query), which will create the table in the database. If the table already exists, the IF NOT EXISTS clause prevents any errors from being thrown. After the execution, the message "Table Bus\_routes created successfully!" will be printed to indicate the success of the operation.

# SQL query to create the table

create\_table\_query = """

CREATE TABLE IF NOT EXISTS Bus\_routes (

ID INT AUTO\_INCREMENT PRIMARY KEY,

Route\_Name TEXT,

Route\_Link TEXT,

Bus\_Name TEXT,

Bus\_Type TEXT,

Departing\_Time TIME,

Duration TEXT,

Reaching\_Time TIME,

Star\_Rating FLOAT,

Price DECIMAL(10, 2),

Seat\_Availability INT

);

"""

cursor.execute(create\_table\_query)

print("Table Bus\_routes created successfully!")

* This code connects to a MySQL database and inserts data from a DataFrame (df) into the Bus\_routes table. It first prepares an SQL INSERT query and then creates a list of data tuples (data\_tuples) where each tuple corresponds to a row in the DataFrame. Using the executemany() function, the data is inserted into the database in bulk. After executing the query, the changes are committed with myconnection.commit(), and the connection is closed, ensuring that the data is successfully saved and the resources are released.

# Insert data into table

insert\_query = """

INSERT INTO Bus\_routes (Route\_Name, Route\_Link, Bus\_Name, Bus\_Type, Departing\_Time, Duration, Reaching\_Time, Star\_Rating, Price, Seat\_Availability)

VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s);

"""

data\_tuples = [

(

row['Route Name'],

row['Route Link'],

row['Bus Name'],

row['Bus Type'],

row['Departure Time'],

row['Duration'],

row['Reaching Time'],

row['Rating'],

row['Price'],

row['Seat\_Availability']

)

for index, row in df.iterrows()

]

# Use executemany for batch inserts

cursor.executemany(insert\_query, data\_tuples)

myconnection.commit()

print("Data inserted successfully!")

cursor.close()

myconnection.close()

**6. Streamlit Application**

**\*** get\_db\_connection() connects to a MySQL database

def get\_db\_connection():

return pymysql.connect(

host='127.0.0.1',

user='root',

password='\*\*\*\*\*\*\*\*',

database='red\_bus'

)

\* load\_data(query) executes an SQL query to retrieve data as a pandas DataFrame. The @st.cache\_data decorator caches the query results for faster performance.

@st.cache\_data

def load\_data(query):

connection = get\_db\_connection()

df = pd.read\_sql(query, connection)

connection.close()

return df

\* This Streamlit code sets up a sidebar for users to filter bus data based on various criteria. Users can select the bus type (e.g., A/C, NON A/C, etc.), enter a route name, adjust the price range, choose a star rating range, and select seat availability through sliders. Once the filters are set, the user can click the "Search" button to apply the filters and display the relevant bus data.

# Sidebar for user input

st.title("Red Bus Data Explorer")

st.write("Explore and filter the data based on the following criteria:")

# Filter page:

st.sidebar.header("Search the Buses")

bus\_type = st.sidebar.selectbox("Select the Bus Type", ["All", "A/C", "NON A/C", "Sleeper", "Semi Sleeper"])

route\_name = st.sidebar.text\_input("Enter the Route Name (leave a blank for all)")

price = st.sidebar.slider("Select the Price range", min\_value=80, max\_value=5000, value=(200, 1000))

star\_rating = st.sidebar.slider("Select the Star Ratings range:", min\_value=1, max\_value=5, value=(1, 2))

seat\_availability = st.sidebar.slider("Select the Seat availability:", min\_value=1, max\_value=65, value=(5, 10))

search = st.sidebar.button("Search")

\* This code dynamically constructs an SQL query to filter bus data based on user input from the sidebar. When the user clicks the "Search" button, the query is built by appending conditions for bus type, route name, price range, star rating, and seat availability. The filters are applied only if the user has specified values for them (e.g., bus type is not "All"). After executing the query, the filtered data is loaded and displayed as a DataFrame in Streamlit. If no buses match the filters, a message is shown, otherwise, the relevant bus data is presented.

# Build SQL Query dynamically

if search:

query = """

SELECT Route\_Name, Bus\_Name, Bus\_Type, Departing\_Time, Duration, Reaching\_Time, Star\_Rating, Price, Seat\_Availability

FROM bus\_routes WHERE 1=1

"""

# Apply filters based on user input

if bus\_type != "All":

query += f" AND Bus\_Type like '%{bus\_type}%'"

if route\_name:

query += f" AND Route\_Name LIKE '%{route\_name}%'"

query += f" AND Price BETWEEN {price[0]} AND {price[1]}"

query += f" AND Star\_Rating BETWEEN {star\_rating[0]} AND {star\_rating[1]}"

query += f" AND Seat\_Availability BETWEEN {seat\_availability[0]} AND {seat\_availability[1]}"

filtered\_data = load\_data(query)

st.write(f"Total no of buses found: {len(filtered\_data)}")

if filtered\_data.empty:

st.write("No buses available based on the selected filters.")

else:

st.dataframe(filtered\_data)

\* This code allows users to view the top 5 rated buses by checking the "Show Top Rated Buses" checkbox in the sidebar. When checked, it executes a SQL query that selects the top 5 buses from the database based on the highest star ratings, ordered in descending order. The result is displayed in a table format on the Streamlit app. If the checkbox is unchecked, the table showing the top-rated buses will disappear, and only the filtered bus data (based on other criteria) will be displayed. By default, the checkbox is checked, so the top-rated buses are shown initially.

# Show Top Rated Buses based on checkbox

if st.sidebar.checkbox("Show Top Rated Buses", value=True):

st.sidebar.write("Uncheck the checkbox to view only filtered bus details.")

top\_rated\_query = """

SELECT Route\_Name, Bus\_Name, Bus\_Type, Star\_Rating, Price

FROM bus\_routes

ORDER BY Star\_Rating DESC

LIMIT 5;

"""

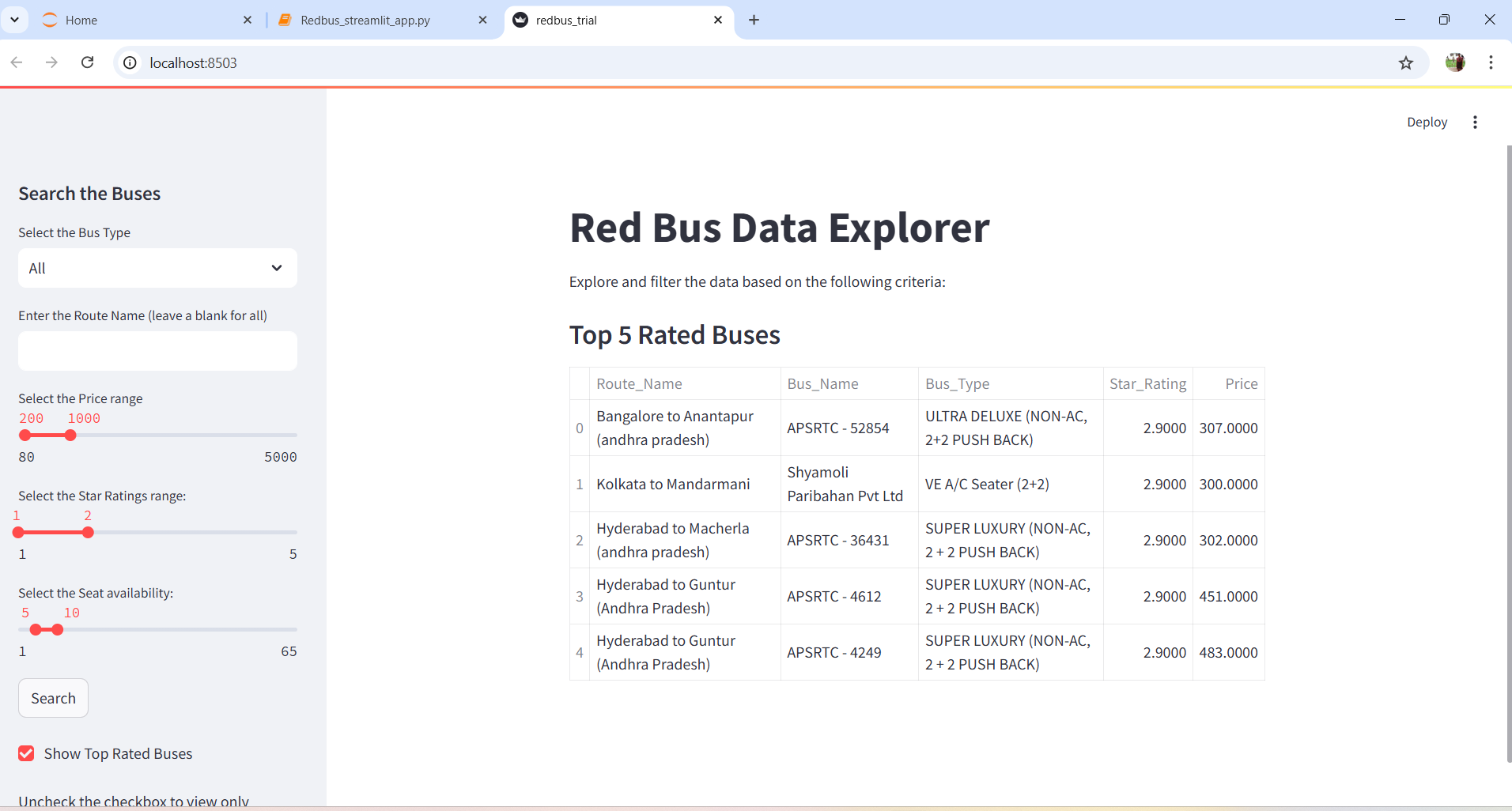
top\_rated\_buses = load\_data(top\_rated\_query)

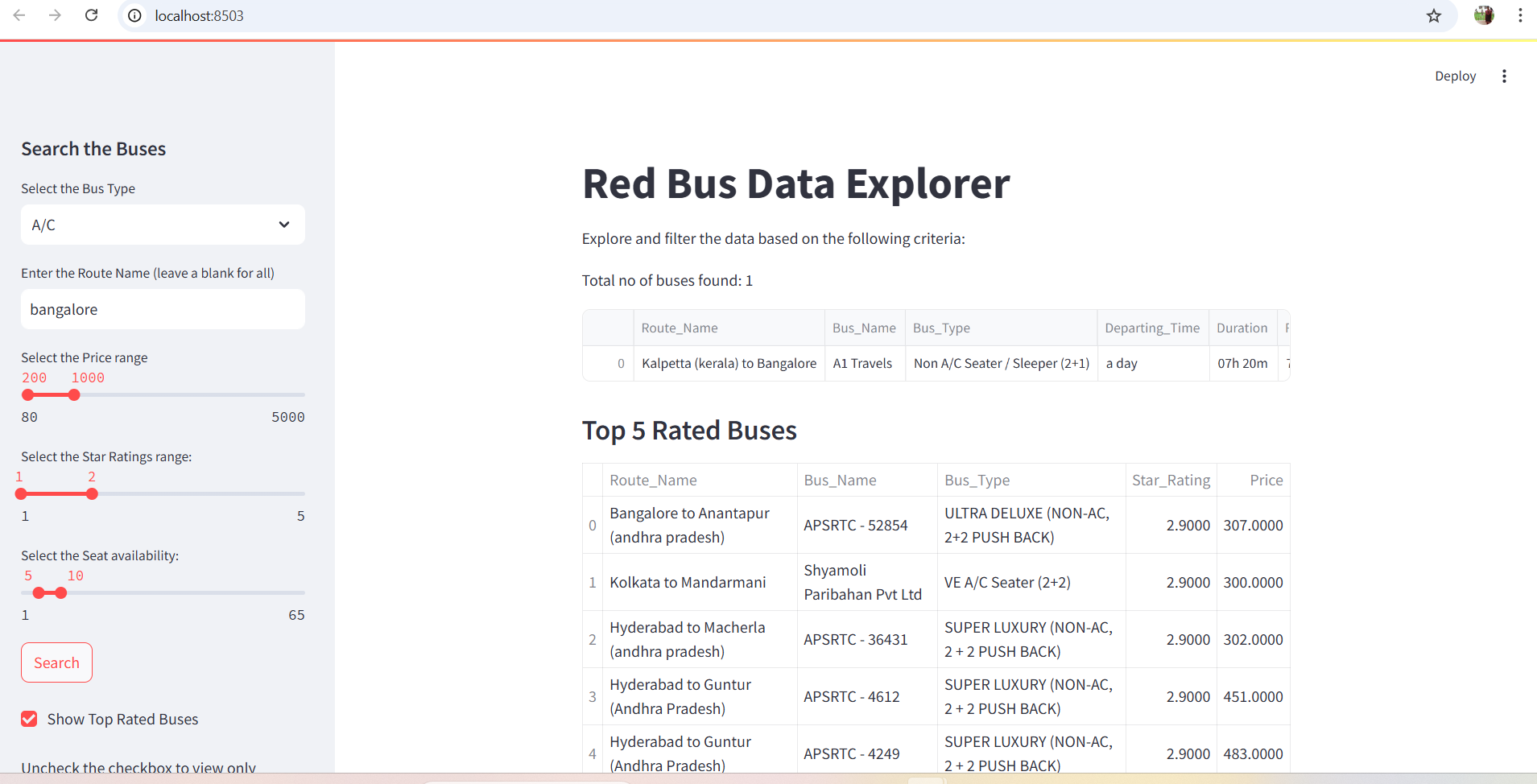
st.subheader("Top 5 Rated Buses")

st.table(top\_rated\_buses)

Streamlit app screenshots:

Streamlit run redbus.py





Local URL: <http://localhost:8503>

Network URL: <http://192.168.0.101:8503>

### **7. Results**

**Expected Outcomes:**

* Successfully scrape a minimum of 10 Government State Bus Transport data from Redbus website using Selenium. Also include the private bus information for the selected routes.
* Store the data in a structured SQL database.
* Develop an interactive Streamlit application for data filtering.
* Ensure the application is user-friendly and efficient.

### **8. Conclusion**

**Summary:** Summarize the project, the process of scraping data, storing it, and displaying it using Streamlit.

**Future Work:**

* Improvements in data scraping.
* Adding more features to the Streamlit app.