Hands-on Lab: CRUD Application Design using Additional Features in Flask

Skills Network

Estimated time needed: **60** minutes

Overview

CRUD, which stands for Create, Read, Update, Delete, are basic functionalities that any application based on a database must possess. The development of these features requires additional knowledge of handling routes and requests. You also require multiple endpoint HTML interfaces to accommodate different requests. The purpose of this lab, therefore, is to give you some additional practice on the usage of Flask and develop a fully functional, CRUD operation-capable web application.

For this lab, you will develop a financial transaction recording system. The system must be capable of **Creating** a new entry, **Reading** existing entries, **Updating** existing entries, and **Deleting** existing entries.

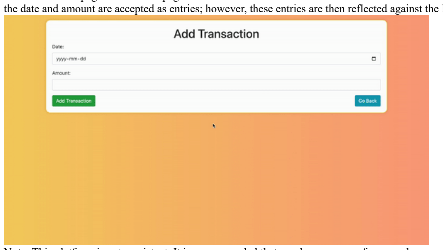
Objectives

After completing this lab, you will be able to:

- Implement "Create" operation to add transaction entry
- Implement "Read" operation to access the list of transaction entries
- Implement "Update" operation to update the details of a given transaction entry
- Implement "Delete" operation to delete a transaction entry.

After you complete developing the application, it will function as displayed in the animation.

The application has three different web pages. The first one displays all the recorded transactions. This page is called Transaction Records and displays all the transactions entries created in the system. This page also gives an option to Edit and Delete the available entries. The option of adding an entry is also available on this page. The second page is Add Transaction which is used when the user chooses to add the entry on the previous page. The user adds the Date and Amount values for the new entry. The third page is Edit Transaction which is user navigated to upon clicking the edit entry option. On this page also, the date and amount are accepted as entries; however, these entries are then reflected against the ID that was being edited.



Note: This platform is not persistent. It is recommended that you keep a copy of your code on your local machines and save changes from time to time. In case you revisit the lab, you will need to recreate the files in this lab environment using the saved copies from your machines.

Let's get started!

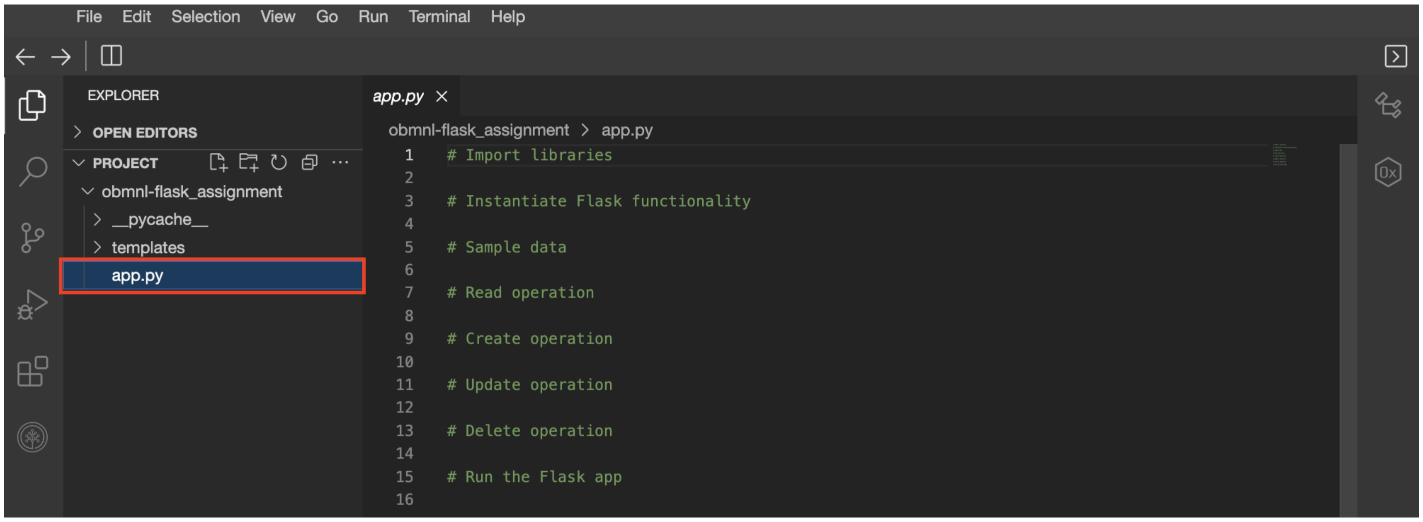
Clone the Project Repository

This lab requires multiple HTML interface files, which have been pre-created for you. You will need to clone the folder structure to the IDE interface using the following command in a terminal shell.

```
git clone https://github.com/ibm-developer-skills-network/obmnl-flask_assignment.git
```

Copied!

When the command is successfully executed, the Project tab must have the folder structure as shown in the image. The root folder, `obmnl-flask_assignment` should have the `templates` folder and a file `app.py`. The `templates` folder has all the required HTML files, `edit.html`, `form.html`, and `transactions.html`. Throughout this lab, you will implement the required functions in `app.py`.



```
# Import libraries

# Instantiate Flask functionality

# Sample data

# Read operation

# Create operation

# Update operation

# Delete operation

# Run the Flask app

```

Initial set up

In the `app.py` file, you need to import necessary modules from Flask and instantiate the Flask application.
For this lab, you will need to import the following functions from the ***flask*** library.

- Flask - to instantiate the application
- request - to process the `GET` and `POST` requests
- url_for - to access the url for a given function using its decorator
- redirect - to redirect access requests according to requirement
- render_template - to render the html page

After importing the functions, instantiate the application to a variable `app`.

▸ *Click here for hint*
▸ *Click here for solution*

Next, let's create a list of sample transactions for testing purposes. You can assume that the transactions already exist on the interface when it is executed for the first time. Please note that this step is completely optional and does not affect the functionality you will develop in this lab. Add the code snippet as shown below to `app.py`.

```
# Sample data
transactions = [
    {'id': 1, 'date': '2023-06-01', 'amount': 100},
    {'id': 2, 'date': '2023-06-02', 'amount': -200},
    {'id': 3, 'date': '2023-06-03', 'amount': 300}
]
```

Copied!

▸ *Click here for solution*

The order in which you will develop the functions is as follows:
1. Read

2. Create
3. Update
4. Delete

The reason to implement **Read** before the other functions is to be able to redirect to the page with all transactions every time a new transaction is created, updated, or deleted. Therefore, the function to read the existing transactions must exist before the others are implemented.

Read Operation

To implement the **Read** operation, you need to implement a route that displays a list of all transactions. This route will handle `GET` requests, which are used to retrieve and display data in `app.py`.

The key steps to implement the Read operation are as follows:

1. Create a function named `get_transactions` that uses `render_template` to return an HTML template named `transactions.html`. This function should pass the transactions to the template for display.

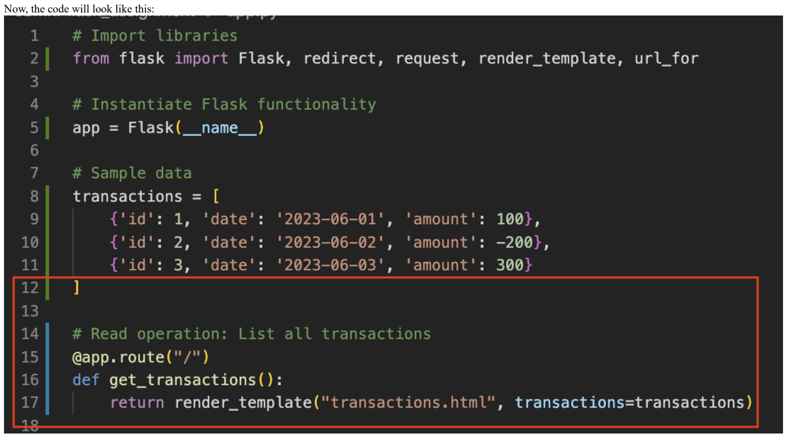
2. Use the Flask `@app.route` decorator to map this function to the root (`/`) URL. This means that when a user visits the base URL of your application, Flask will execute the `get_transactions` function and return its result.

▸ *Click here for hint*
▾ *Click here for solution*

```
# Read operation: List all transactions
@app.route("/")
def get_transactions():
    return render_template("transactions.html", transactions=transactions)
```

Copied!

Now, the code will look like this:



```
# Import libraries
from flask import Flask, redirect, request, render_template, url_for

# Instantiate Flask functionality
app = Flask(__name__)

# Sample data
transactions = [
    {'id': 1, 'date': '2023-06-01', 'amount': 100},
    {'id': 2, 'date': '2023-06-02', 'amount': -200},
    {'id': 3, 'date': '2023-06-03', 'amount': 300}
]

# Read operation: List all transactions
@app.route("/")
def get_transactions():
    return render_template("transactions.html", transactions=transactions)
```

Create Operation

For the **Create** operation, you will implement a route that allows users to add new transactions. This will involve handling both `GET` and `POST` HTTP requests - `GET` for displaying the form to the user and `POST` for processing the form data sent by the user.

Here is the list of steps to implement the Create operation.

1. Create a function named `add_transaction`.

2. Use `add` as the decorator for this function. Make sure to pass both `GET` and `POST` as possible methods.

3. If the request method is `GET`, use the `render_template` function to display an HTML form using a template named `form.html`. This form will allow users to input data for a new transaction.

4. If the request method is `POST`, use `request.form` to extract the form data, create a new transaction, append it to the transactions list, and then use `redirect` and `url_for` to send the user back to the list of transactions.

5. The new transaction is passed on to the reading function in the following format.

```
transation = {
              'id': len(transactions)+1
              'date': request.form['date']
              'amount': float(request.form['amount'])
             }
```

Copied!

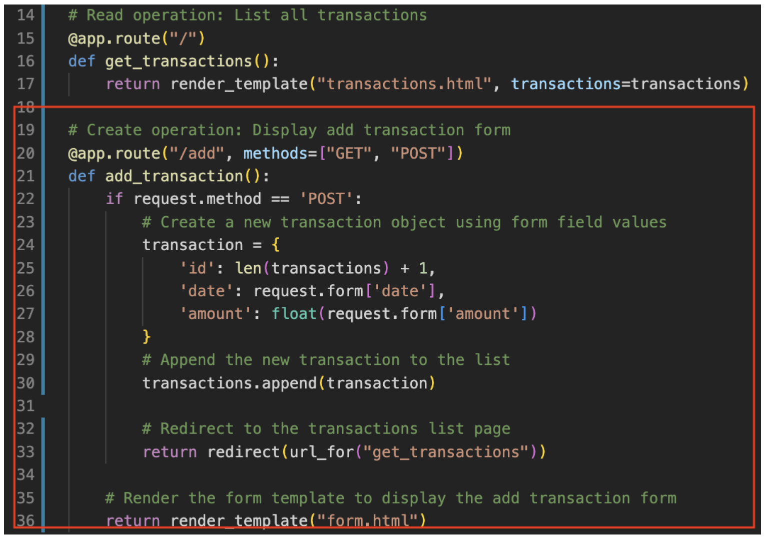
Here, request.form function parses the information received from the entry made in the form.

▸ *Click here for hint*
▾ *Click here for solution*

```
# Create operation: Display add transaction form
# Route to handle the creation of a new transaction
@app.route("/add", methods=["GET", "POST"])
def add_transaction():
    # Check if the request method is POST (form submission)
    if request.method == 'POST':
        # Create a new transaction object using form field values
        transaction = {
            'id': len(transactions) + 1,            # Generate a new ID based on the current length of the transactions list
            'date': request.form['date'],           # Get the 'date' field value from the form
            'amount': float(request.form['amount']) # Get the 'amount' field value from the form and convert it to a float
        }
        # Append the new transaction to the transactions list
        transactions.append(transaction)

        # Redirect to the transactions list page after adding the new transaction
        return redirect(url_for("get_transactions"))

    # If the request method is GET, render the form template to display the add transaction form
    return render_template("form.html")
```

Copied!

Now, the code will look like this:



```python
# Read operation: List all transactions
@app.route("/")
def get_transactions():
    return render_template("transactions.html", transactions=transactions)

# Create operation: Display add transaction form
@app.route("/add", methods=["GET", "POST"])
def add_transaction():
    if request.method == 'POST':
        # Create a new transaction object using form field values
        transaction = {
            'id': len(transactions) + 1,
            'date': request.form['date'],
            'amount': float(request.form['amount'])
        }
        # Append the new transaction to the list
        transactions.append(transaction)

        # Redirect to the transactions list page
        return redirect(url_for("get_transactions"))

    # Render the form template to display the add transaction form
    return render_template("form.html")
```

Note: The statements outside the `if` case are, by default, the `else` case. The statements in the `if` case end with a return statement; hence only one of the two cases will run at a time.

Update Operation

For the **Update** operation, you need to implement a route that allows users to update existing transactions. You'll again handle both `GET` and `POST` HTTP requests - `GET` for displaying the current transaction data in a form, and `POST` for processing the updated data sent by the user.

Complete the following steps to implement the Update operation.

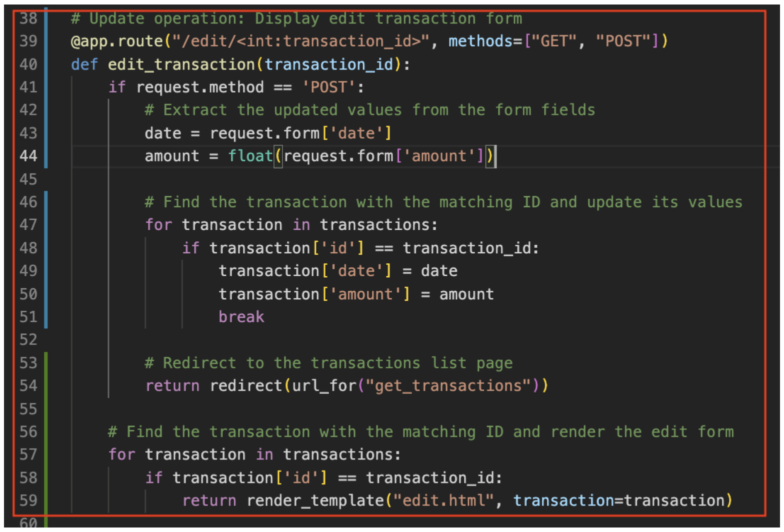
1. Create a function named `edit_transaction` that handles both `GET` and `POST` requests. This function should accept a parameter, `transaction_id`.
2. Decorate the function with `@app.route` and use the route string `/edit/<int:transaction_id>`. The `<int:transaction_id>` part in the URL is a placeholder for any integer. Flask will pass this integer to your function as the `transaction_id` argument.
3. If the request method is `GET`, find the transaction with the ID that matches `transaction_id` and use `render_template` to display a form pre-populated with the current data of the transaction using a template named `edit.html`.
4. If the request method is `POST`, use request.form to get the updated data, find the transaction with the ID that matches `transaction_id` and modify its data, then redirect the user back to the list of transactions.

▾ *Click here for solution*

```python
# Update operation: Display edit transaction form
# Route to handle the editing of an existing transaction
@app.route("/edit/<int:transaction_id>", methods=["GET", "POST"])
def edit_transaction(transaction_id):
    # Check if the request method is POST (form submission)
    if request.method == 'POST':
        # Extract the updated values from the form fields
        date = request.form['date']           # Get the 'date' field value from the form
        amount = float(request.form['amount'])# Get the 'amount' field value from the form and convert it to a float

        # Find the transaction with the matching ID and update its values
        for transaction in transactions:
            if transaction['id'] == transaction_id:
                transaction['date'] = date       # Update the 'date' field of the transaction
                transaction['amount'] = amount   # Update the 'amount' field of the transaction
                break                            # Exit the loop once the transaction is found and updated

        # Redirect to the transactions list page after updating the transaction
        return redirect(url_for("get_transactions"))

    # If the request method is GET, find the transaction with the matching ID and render the edit form
    for transaction in transactions:
        if transaction['id'] == transaction_id:
            # Render the edit form template and pass the transaction to be edited
            return render_template("edit.html", transaction=transaction)

    # If the transaction with the specified ID is not found, handle this case (optional)
    return {"message": "Transaction not found"}, 404
```

Copied!

Now, the code will look like this:



```python
# Update operation: Display edit transaction form
@app.route("/edit/<int:transaction_id>", methods=["GET", "POST"])
def edit_transaction(transaction_id):
    if request.method == 'POST':
        # Extract the updated values from the form fields
        date = request.form['date']
        amount = float(request.form['amount'])

        # Find the transaction with the matching ID and update its values
        for transaction in transactions:
            if transaction['id'] == transaction_id:
                transaction['date'] = date
                transaction['amount'] = amount
                break

        # Redirect to the transactions list page
        return redirect(url_for("get_transactions"))

    # Find the transaction with the matching ID and render the edit form
    for transaction in transactions:
        if transaction['id'] == transaction_id:
            return render_template("edit.html", transaction=transaction)
```

Note: There may be multiple ways of achieving the same result. Please use the solution given above only as a reference.

Delete Operation

Finally, you need to implement a route that allows users to delete existing transactions.

Complete the following steps to implement the Delete operation.

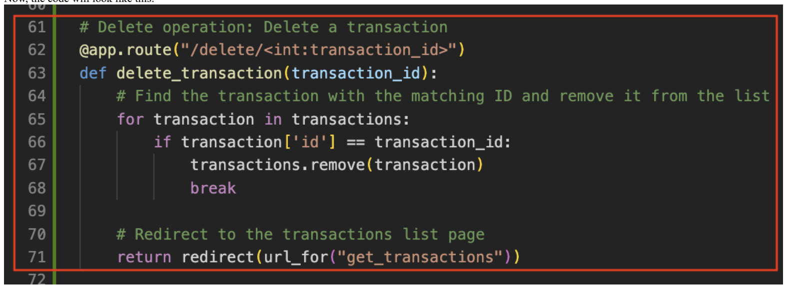
1. Create a function named `delete_transaction` that takes a parameter, `transaction_id`.
2. Decorate the function with `@app.route` and use the route string `/delete/<int:transaction_id>`. The `<int:transaction_id>` part in the URL is a placeholder for any integer. Flask will pass this integer to your function as the `transaction_id` argument.
3. In the function body, find the transaction with the ID that matches `transaction_id` and remove it from the transactions list, then `redirect` the user back to the list of transactions.

▾ *Click here for solution*

```python
# Delete operation: Delete a transaction
# Route to handle the deletion of an existing transaction
@app.route("/delete/<int:transaction_id>")
def delete_transaction(transaction_id):
    # Find the transaction with the matching ID and remove it from the list
    for transaction in transactions:
        if transaction['id'] == transaction_id:
            transactions.remove(transaction)  # Remove the transaction from the transactions list
            break  # Exit the loop once the transaction is found and removed

    # Redirect to the transactions list page after deleting the transaction
    return redirect(url_for("get_transactions"))
```

Copied!

Now, the code will look like this:



```python
# Delete operation: Delete a transaction
@app.route("/delete/<int:transaction_id>")
def delete_transaction(transaction_id):
    # Find the transaction with the matching ID and remove it from the list
    for transaction in transactions:
        if transaction['id'] == transaction_id:
            transactions.remove(transaction)
            break

    # Redirect to the transactions list page
    return redirect(url_for("get_transactions"))
```

Finishing Steps and Running the Application

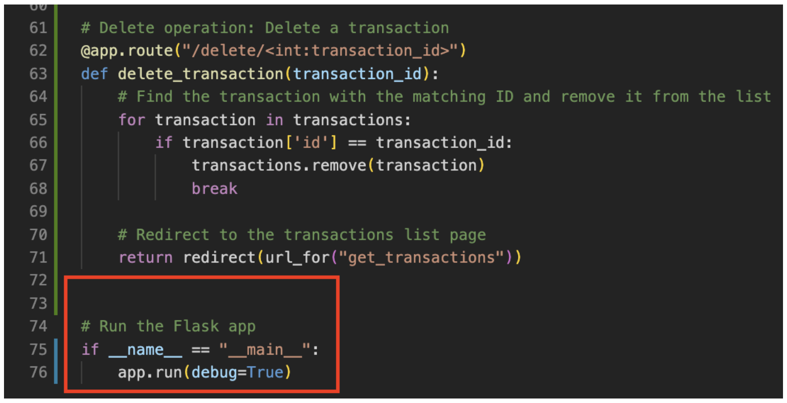
Check if the current script is the main program (that is, it wasn't imported from another script) with the conditional `if __name__ == "__main__":`.

If the condition is true, call `app.run(debug=True)` to start the Flask development server with debug mode enabled. This will allow you to view detailed error messages in your browser if something goes wrong.

```python
# Run the Flask app
if __name__ == "__main__":
    app.run(debug=True)
```

Copied!

Now, the code will look like this:



```
# Delete operation: Delete a transaction
@app.route("/delete/<int:transaction_id>")
def delete_transaction(transaction_id):
    # Find the transaction with the matching ID and remove it from the list
    for transaction in transactions:
        if transaction['id'] == transaction_id:
            transactions.remove(transaction)
            break

    # Redirect to the transactions list page
    return redirect(url_for("get_transactions"))


# Run the Flask app
if __name__ == "__main__":
    app.run(debug=True)
```

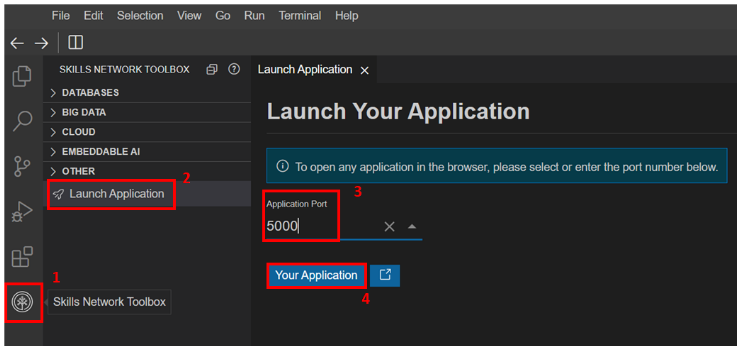
The code is now complete. Run the file `app.py` from a terminal shell using the command:

```
1. 1
1. python3.11 app.py
```

Copied! Executed!

By default, Flask launches the application on LocalHost:5000. As displayed in the image,

1. Launch the application by going to the Skills Network Library, going to `Launch Application`.
2. Enter `5000` in the port number and launch the application window.

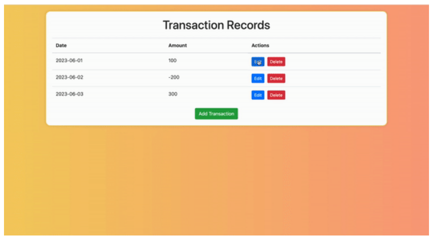


Launch Your Application

To open any application in the browser, please select or enter the port number below.

Application Port: 5000

The final application looks like this.



Transaction Records

Date	Amount	Actions
2023-06-01	100	Edit Delete
2023-06-02	-200	Edit Delete
2023-06-03	300	Edit Delete

Lab Help

In case you face an error while going through all the steps, the final code for `app.py` is being shared here as a ready reference. Please note that this should be used only as a last resort to ensure that you gain the learning intended through this lab.

▸ *Final code for app.py*

Testing the Interface

Once your application is ready, try the CRUD operations on the launched application. Possible tasks for testing the application could be:

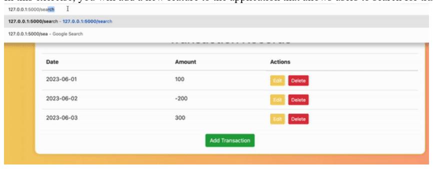
1. Click on the "Add" button to open the form and add a new transaction.
2. Click on the "Edit" button for any transaction and update the information (date and amount) for the transaction.
3. Click on the "Delete" button for any transaction to delete it from the list.
4. Verify the transactions are displayed correctly.

Practice Exercises

The following are some practice exercises for the interested learners. We are not providing the solutions for these exercises to encourage the learners to try them on their own. Please feel free to use the course discussion forum for sharing your opinions on the solution with other interested learners.

Exercise 1: Search Transactions

In this exercise, you will add a new feature to the application that allows users to search for transactions within a specified amount range. You will create a new route called `/search` that handles both `GET` and `POST` requests in `app.py`.



127.0.0.1:5000/search
127.0.0.1:5000/search - 127.0.0.1:5000/search
127.0.0.1:5000/sea - Google Search

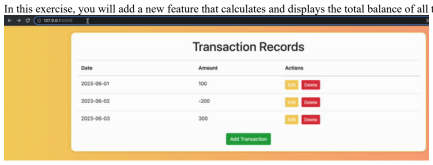
Date	Amount	Actions
2023-06-01	100	Edit Delete
2023-06-02	-200	Edit Delete
2023-06-03	300	Edit Delete

Instructions:

1. Create a new function named `search_transactions` and use the `@app.route` decorator to map it to the URL `/search`.
2. Inside the function, check if the request method is `POST`. If it is, retrieve the minimum and maximum amount values from the form data submitted by the user. Convert these values to floating-point numbers.
3. Filter the transactions list based on the amount range specified by the user. Create a new list, `filtered_transactions`, that contains only the transactions whose amount falls within the specified range. You can use a list comprehension for this.
4. Pass the `filtered_transactions` list to the `transactions.html` template using the `render_template` function. In this template, display the transactions similar to the existing `transactions.html` template.
5. If the request method is `GET`, render a new template called `search.html`. This template should contain a form that allows users to input the minimum and maximum amount values for the search.

Exercise 2: Total Balance

In this exercise, you will add a new feature that calculates and displays the total balance of all transactions. You will create the route in `app.py`.



Transaction Records

Date	Amount	Actions
2023-06-01	100	Edit Delete
2023-06-02	-200	Edit Delete
2023-06-03	300	Edit Delete

Instructions:

1. Create a new function named `total_balance` and use the `@app.route` decorator to map it to the URL `/balance`.
2. Inside the function, calculate the total balance by summing the amount values of all transactions in the transactions list.
3. Return the total balance as a string in the format “Total Balance: {balance}”.
4. To display the total balance, you do not need to create a new template. Instead, you will modify the `transactions.html` template to include the total balance value at the bottom of the table.
5. After displaying the list of transactions in the `transactions.html` template, add a new row to display the total balance. You can use the same `render_template` function as before, passing both the transactions list and the total balance value.

Conclusion

Congratulations on completing this lab.

In this lab, you have learned how to:

- Implement CRUD functionality in a database application.
- Use additional functions from Flask library for advanced routing and request management.
- Manage routing between multiple HTML files as per requirement.

Author(s)

Vicky Kuo

Additional Contributor

Abhishek Gagneja

Changelog

Date	Version	Changed by	Change Description
2023-07-24	2.0	Steve Hord	QA pass with edits
2023-07-15	1.0	Vicky Kuo	Initial version created