CRUD Operations using Additional Features in Flask

Estimated time needed: 15 minutes

Overview

Create, Read, Update, and **Delete** (CRUD) are basic functions that any application with a database must perform. To effectively implement the CRUD operations, you will need to manage different HTTP methods, such as `GET` and `POST` requests. A `GET` request is generally used to retrieve or read data and is often used to display a form. A `POST` request is commonly used to send data to create or update data. An example of a POST request is form submission.

This reading aims to introduce you to additional features of Flask, focusing on the CRUD functions. You will also learn how these functions are interrelated and how they utilize the different HTML files, dynamic routes, and various HTTP methods.

Objectives

In this reading, you will:

- **Access form data** to capture user inputs with `flask.request.form` in `POST` requests
- Control user navigation: using the Flask's `redirect` function
- **Generate URLs dynamically** using `url_for` to create adaptable URLs in your Flask application
- **Manage different HTTP request types** to design flexible routes that respond to various HTTP request types
- **Implement CRUD operations** for data management in a Flask app

Note: Each section includes relevant code examples and explanations to enhance your understanding of the crucial Flask features.

Accessing Form Data with flask.request.form

You can use `flask.request.form` to access form data that a user has submitted via a `POST` request. For instance, this feature can be used if you have a login form with username and password fields.

In your HTML file, you might have a form like this:

```
<form method="POST" action="/login">
    <input type="text" name="username">
    <input type="password" name="password">
    <input type="submit" value="Submit">
</form>
```

The Python code to access the username and password will be as follows:

```
from flask import request
@app.route('/login', methods=['POST'])
def login():
    username = request.form['username']
    password = request.form['password']
    # process login here
```

Redirecting to a URL with flask.redirect

Flask provides a function called flask.redirect to guide users to a different webpages (or endpoints). The flask.redirect function can be useful in several scenarios. For example, you can use the flask.redirect function to redirect a user to a **login** page when they try to access a restricted **admin** page.

Python code:

```
from flask import redirect
@app.route('/admin')
def admin():
    return redirect('/login')
```

Generating Dynamic URLs with flask.url_for

The `flask.url_for` function dynamically generates URLs for a given endpoint. Dynamically generating URLs can be particularly useful when the URL for a route is altered. The `flask.url_for` function automatically updates the URL throughout your templates or code, minimizing manual work. For example, consider the scenario where a user is trying to access the **admin** page and must be redirected to the **login** page. In this scenario, `url_for('login')` will retrieve the URL for the **login** page from the existing routes.

Python code:

```
from flask import url_for
@app.route('/admin')
def admin():
    return redirect(url_for('login'))
@app.route('/login')
def login():
    return "<Login Page>"
```

Handling different HTTP request types

Flask allows you to define routes to manage different types of HTTP requests. You can define the route with both the access methods, GET and POST, and in the function description, define the use cases for both methods.

Python code:

```
@app.route('/data', methods=['GET', 'POST'])
def data():
    if request.method == 'POST':
        # process POST request
    if request.method == 'GET':
        # process GET request
```

In the HTML file, you will add a form that allows both `GET` and `POST` requests:

```
<!-- For POST -->
<form method="POST" action="/data">
    <!-- Your input fields here -->
    <input type="submit" value="Submit">
</form>
<!-- For GET -->
<a href="/data">Fetch data</a>
```

In the last example, the `/data` route accepts both `GET` and `POST` requests. The type of the request can be checked using `flask.request.method`.

CRUD operations

The CRUD operations represent the four basic functions that you require to interact with any persistent storage, such as a database. In web development, the CRUD operations often correspond to HTTP methods.

Create operation

Creating data often involves presenting a form to the user to gather the information that you want to store in the database as a new record. In Flask, this data is accessed using `flask.request.form`.

HTML form for creating data:

```
<form method="POST" action="/create">
    <input type="text" name="name">
    <input type="submit" value="Create">
</form>
```

Python code:

```
@app.route('/create', methods=['GET', 'POST'])
def create():
    if request.method == 'POST':
        # Access form data
        name = request.form['name']
        # Create a new record with the name
        record = create_new_record(name)  # Assuming you have this function defined
        # Redirect user to the new record
        return redirect(url_for('read', id=record.id))
    # Render the form for GET request
    return render_template('create.html')
```

Read operation

Reading data involves accessing the data and presenting it to the user. To access specific entries, the request needs to go with specific IDs. Therefore, you will need to pass the ID as an argument to the function. The following example shows that the ID can be accessed from the route.

Python code:

```
@app.route('/read/<int:id>', methods=['GET'])
def read(id):
    # Get the record by id
    record = get_record(id)  # Assuming you have this function defined
    # Render a template with the record
    return render_template('read.html', record=record)
```

Update operation

Updating data requires the process of accessing specific entries, like the **Read** operation, and involves giving new data to the concerned parameter, like the **Create** operation. Therefore, the route should access the ID and contain both access methods.

Sample HTML form for updating data:

```
<form method="POST" action="/update/{{record.id}}">
    <input type="text" name="name" value="{{record.name}}">
    <input type="submit" value="Update">
</form>
```

Python code:

```
@app.route('/update/<int:id>', methods=['GET', 'POST'])
def update(id):
    if request.method == 'POST':
        # Access form data
        name = request.form['name']
        # Update the record with the new name
        update_record(id, name)  # Assuming you have this function defined
        # Redirect user to the updated record
        return redirect(url_for('read', id=id))

    # Render the form for GET request with current data
    record = get_record(id)  # Assuming you have this function defined
    return render_template('update.html', record=record)
```

Delete operation

Deleting data involves removing a record based on its ID. The Delete operation will typically require the ID to be passed, as reported by the HTML page, in the form of an argument to the function.

Sample HTML form for deleting data:

```
<form method="POST" action="/delete/{{record.id}}">
    <input type="submit" value="Delete">
</form>
```

Python code:

```
@app.route('/delete/<int:id>', methods=['POST'])
def delete(id):
    # Delete the record
    delete_record(id)  # Assuming you have this function defined
    # Redirect user to the homepage
    return redirect(url_for('home'))
```

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
2023-07-20	1.0	Vicky Kuo	Initial version created