Quick Cart

Web Services

<Version 1.0.1>

Prepared by

Dejeon Battick	**180914**	**dejeon.battick11@gmail.com**
Shemar Lundy	**180916**	**lundyshemar@yahoo.com**
Shemar Henry	**180915**	**shemarhenry24@yahoo.com**
Mark-Anthony Jones	**180920**	**jmarkanthony.062@gmail.com**

Course Instructor: Thomas Xu

Course: Web and Mobile Application development II

Date: June 23, 2019

Overall Description

Product Context and Need

The Quick Cart Blog was conceptualised with the average shopper in mind. The blog works alongside the Quick Cart mobile application for android and iOS and allows the user to have a sense of community. This community will allow users to share recipes with each other and be kept abreast on posts made by the Quick Cart blog administrator. These posts maybe but are not limited, sales, specials etc.

Product Functionality-

Major functions of the system will be to:

1. Allows the admin user to make posts to the blog.
2. Allows all users to add recipes to the blog.
3. Allows a user to delete their post and or recipe.
4. Allows a user to update their post and or recipes.
5. Allows a user view all their posts and or recipes.

Non-functional requirements of the system will be to:

1. Posts and recipes should be stored on a secured database.
2. User password should be hashed to ensure security.
3. System must check credentials against the user database to ensure that there isn't duplication in the email field.
4. System should check to ensure that form fields are properly filled out.

5. System should ensure that a user is logged in before accessing certain features such as creating, updating and deleting posts and or recipes.
6. Should have separate dashboards for administrative users and regular users.

Stakeholders and Users Characteristics

The key stakeholders of this system are:

Shoppers:

- Persons who use the Quick Cart application and also utilise the blog feature.

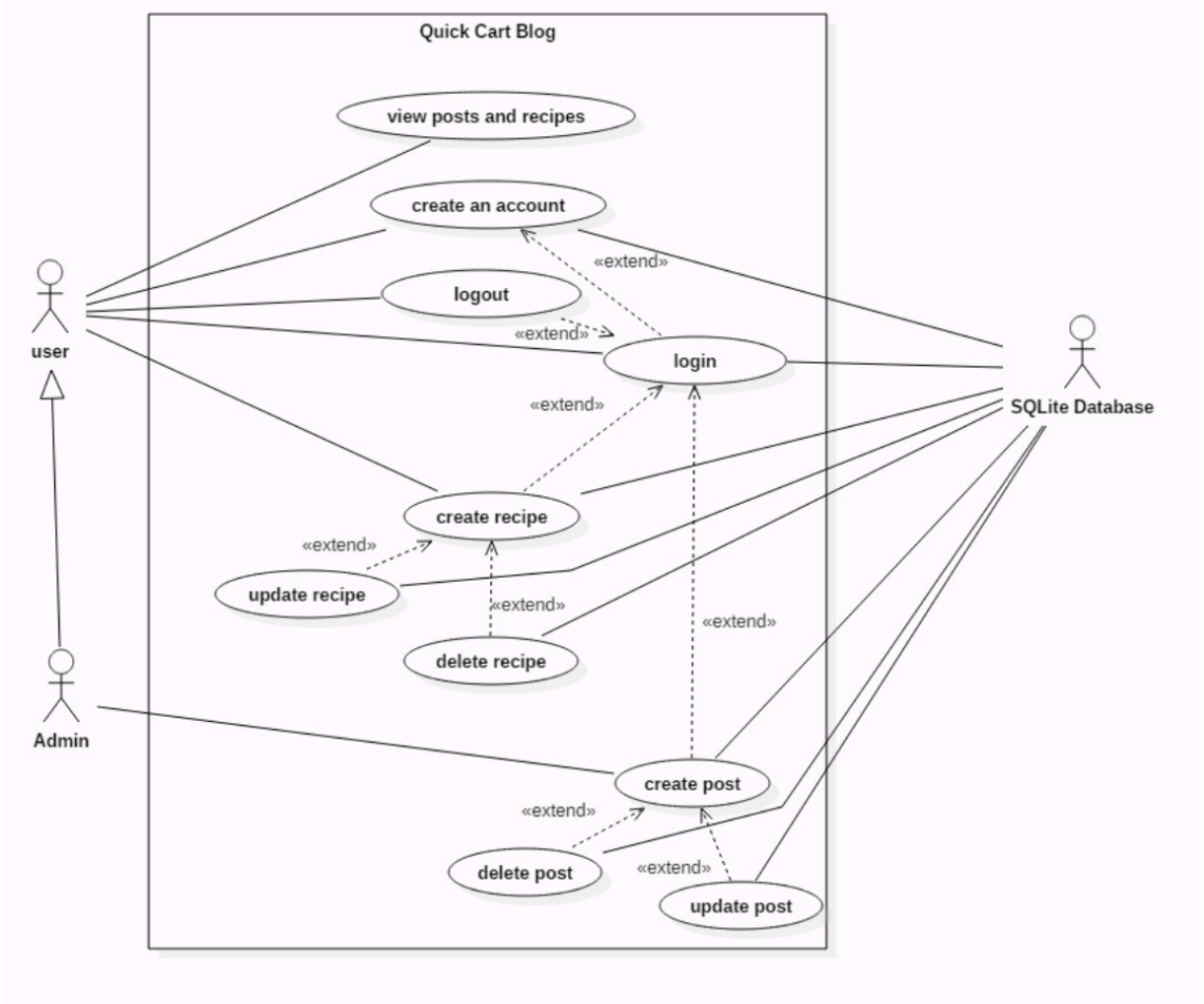
Store Staff:

- Persons who make posts on the behalf of the Store through the use of the administrative login.

Operating Environment

Seeing as the blog is a Flask web service an internet connection is needed in order to access the site. The server on which the web service is running must be up and running and publicly accessible.

Use Case Diagram - Blog



Use Case Description

Use case name: view posts and recipes

Summary: the actor should be able to view all the posts and recipes that have been created.

Primary Actor: user, admin

Secondary Actor: n/a

Precondition: the user must access the site to view all the posts and recipes.

Main Sequence:

1. The user accesses the blog through use of the app.
2. The user is brought to the home page where they can view all the posts made.
3. The system adds the party to the database.

Alternative sequence:

Step 2: if the user clicks the recipe link, he/she will be brought to the recipe page where they can view all the recipes that have been added to the blog.

Post-condition: the posts and recipes added to the system have been successfully viewed by the user.

Use case name: create an account

Summary: the actor should be able to create an account for the blog.

Primary Actor: user, admin

Secondary Actor: database

Precondition: the site was successfully launched and the register link was clicked.

Main Sequence:

1. The user fills out all the necessary field in the registration form.
2. The user clicks the submit button.
3. The site validates the information that has been submitted by the user.
4. The user is brought to the login page.

Alternative sequence:

Step 3: if the information submitted does not pass the validation the creation process will fail and the user will be prompted with the necessary error message(s).

Post-condition: the user has successfully created an account.

Use case name: login

Summary: the actor should be able to login to the site using the credentials they provide when creating the account.

Primary Actor: user, admin

Precondition: site was successfully launched and the user is on the login page.

Main Sequence:

1. User provides the required information.
2. The site validates the given information.
3. The site then brings the user to the home page and allows access to login specific features.

Alternative sequence:

Step 2: if the information submitted does not pass the validation check the login process will fail and the user will be prompted with the necessary error message(s).

Post-condition: the user was successfully logged into their account.

Use case name: logout

Summary: the actor should be able to logout to the site.

Primary Actor: user, admin

Precondition: the site was successfully launched and the user is currently logged in.

Main Sequence:

1. User clicks the logout button.

2. The site logs out the user and brings them back to the log in page.

Post-condition: the user was successfully logged out of their account.

Use case name: create recipe

Summary: the actor should be able to create a recipe post for the blog.

Primary Actor: user, admin

Secondary Actor: database

Precondition: the site was successfully launched and the actor successfully logged in.

Main Sequence:

1. The actor fills out all the necessary field in the create recipe form.
2. The actor clicks the submit button.
3. The site validates the information that has been submitted by the actor.
4. The actor is brought to the recipe page.

Alternative sequence:

Step 3: if the information submitted does not pass the validation check the recipe posting process will fail and the user will be prompted with the necessary error message(s).

Post-condition: the user has successfully created a recipe post.

Use case name: delete recipe

Summary: the actor should be able to delete his/her recipe post for the blog.

Primary Actor: user, admin

Secondary Actor: database

Precondition: the site was successfully launched, the user logged in and a recipe is exists in the database.

Main Sequence:

1. The actor goes to the account page.
2. The actor clicks the delete button for a recipe.
3. The recipe is deleted.

Post-condition: the user has successfully deleted their recipe post.

Use case name: update recipe

Summary: the actor should be able to update his/her recipe post for the blog.

Primary Actor: user, admin

Secondary Actor: database

Precondition: the site was successfully launched, the user logged in and a recipe exists in the database.

Main Sequence:

1. The actor goes to the account page.
2. The actor clicks the update button for a recipe.
3. The actor is then brought to the update recipe page where the recipe information is displayed in the form.
4. The actor edits the displayed information.
5. The actor clicks the submit button.
6. The site does a validation check on the newly submitted information.
7. The actor is brought to the recipe page.

Alternative sequence:

Step 6: if the information submitted does not pass the validation check the recipe updating process will fail and the actor will be prompted with the necessary error message(s).

Post-condition: the actor has successfully updated their recipe post.

Use case name: create post

Summary: the actor should be able to create a post for the blog.

Primary Actor: admin

Secondary Actor: database

Precondition: the site was successfully launched and the actor successfully logged in.

Main Sequence:

1. The actor fills out all the necessary field in the create post form.
2. The actor clicks the submit button.
3. The site validates the information that has been submitted by the actor.
4. The actor is brought to the home page.

Alternative sequence:

Step 3: if the information submitted does not pass the validation check the posting process will fail and the user will be prompted with the necessary error message(s).

Post-condition: the user has successfully created a post.

Use case name: delete post

Summary: the actor should be able to delete his/her post for the blog.

Primary Actor: admin

Secondary Actor: database

Precondition: the site was successfully launched, the user logged in and a post exists in the database.

Main Sequence:

4. The actor goes to the account page.
5. The actor clicks the delete button for a post.
6. The post is deleted.

Post-condition: the user has successfully deleted their post.

Use case name: update post

Summary: the actor should be able to update his/her post for the blog.

Primary Actor: admin

Secondary Actor: database

Precondition: the site was successfully launched, the user logged in and a post exists in the database.

Main Sequence:

1. The actor goes to the account page.
2. The actor clicks the update button for a post.
3. The actor is then brought to the update post page where the post details are displayed in the form.
4. The actor edits the displayed information.
5. The actor clicks the submit button.

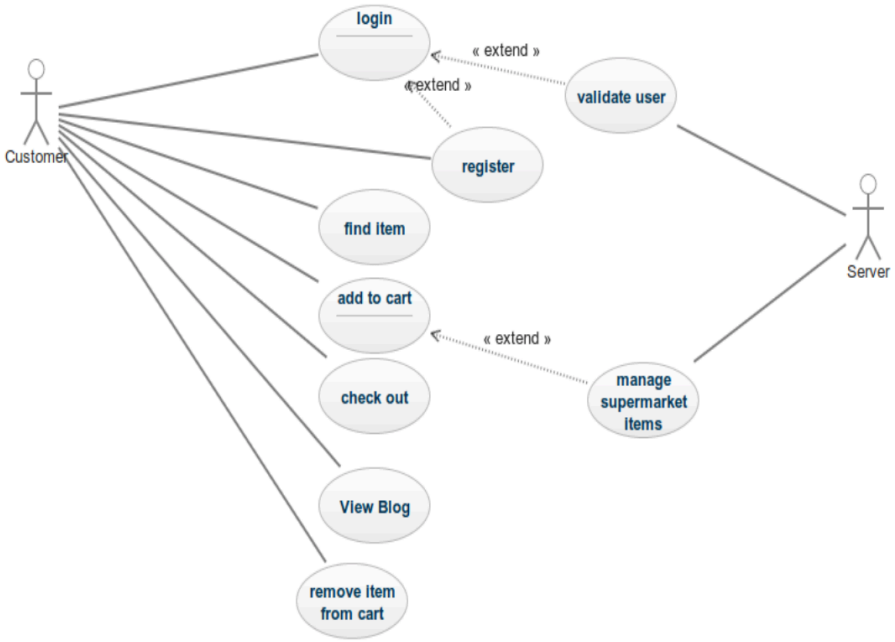
6. The site does a validation check on the newly submitted information.
7. The actor is brought to the home page.

Alternative sequence:

Step 6: if the information submitted does not pass the validation check the recipe updating process will fail and the actor will be prompted with the necessary error message(s).

Post-condition: the actor has successfully updated their recipe post.

Use Case Diagram - Server



Use Case Description

Use case name: validate user

Summary: the server must be able to validate user login information on login request

Primary Actor: server

Secondary Actor: customer

Precondition: a user tried to login to mobile application

Main Sequence:

1. The server compares entered username and password to existing user information
2. If successful, the server responds with a JSON response that would contain other useful user information.

Alternative sequence:

Step 2: If the information submitted does not pass the validation check the server response with a JSON response containing an appropriate error.

Post-condition: the user successfully logs into application

Use case name: manage supermarket items

Summary: the server must be able to keep a collection of all items in supermarket, and upon request return a list of appropriate items matching request to client.

Primary Actor: server

Secondary Actor: customer

Precondition: a user makes a search request

Main Sequence:

1. The server compares entered search to existing item tags
2. Items with tags that match the search request are added to a list
3. That list is converted to a JSONlist and appended to a JSONarray matched by the search request

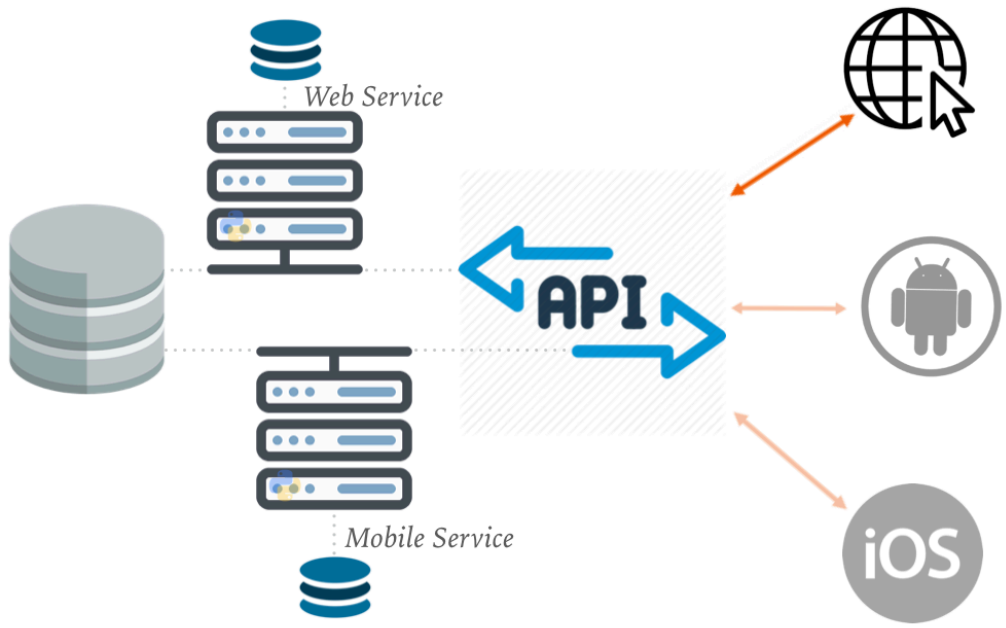
4. The JSONarray is then returned to the client

Alternative sequence:

Step 2: If there are no matching items appropriate JSOn error response is returned to client instead

Post-condition: the user gets list of available search related items

Structure



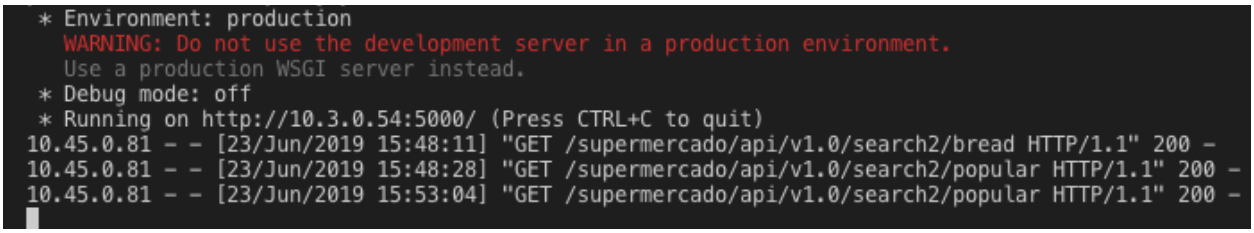
Blog

The Quick Cart Blog was developed using the Flask API. Flask is a restful API that is used in python that allows for developers to create web services that run on a server that was created in python. In implementing the functionality of the blog, several features of flask was utilised.

WTForms were used to create the forms that captured user input and added them to the database. Along with these forms Validators were used to ensure that user input was appropriate and did not contain anything that would be fatal to the web service.

Server

The server uses Flask, SQL-Achemy and SQL-Marshmellow to manage a MySQL Database. The Database stores all user and supermarket item information, which the Flask based python server accesses and manipulates in the form of a REST API.



```python
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField, BooleanField, TextAreaField
from wtforms.validators import DataRequired, Length, Email, EqualTo, ValidationError
from QCblog.models import User

class RegistrationForm(FlaskForm):
    username= StringField('Username', validators=[DataRequired(), Length(min=2, max=20)])
    email= StringField('Email', validators=[DataRequired(), Email()])
    password= PasswordField('Password', validators=[DataRequired()])
    confirm_password= PasswordField('Confirm Password', validators=[DataRequired(), EqualTo('password')])
    submit= SubmitField('Sign up')

    def validate_username(self, username):
        user = User.query.filter_by(username=username.data).first()
        if user:
            raise ValidationError('Sorry that name is taken.')

    def validate_email(self, email):
        user = User.query.filter_by(email=email.data).first()
        if user:
            raise ValidationError('This email is already in use.')
class LoginForm(FlaskForm):
    email= StringField('Email', validators=[DataRequired(), Email()])
    password= PasswordField('Password', validators=[DataRequired()])
    remember= BooleanField('Remember Me')
    submit= SubmitField('Login')

class RecipeForm(FlaskForm):
    title= StringField('Title', validators=[DataRequired(), Length(min=2, max=20)])
    recipe= TextAreaField('Recipe', validators=[DataRequired()])
    submit= SubmitField('Add Recipe')

class PostForm(FlaskForm):
    title= StringField('Title', validators=[DataRequired(), Length(min=2, max=20)])
    post= TextAreaField('Post', validators=[DataRequired()])
    submit= SubmitField('Create Post')
```

The database was created using SQLAlchemy which always flask to communicate with databases that can be utilised by the web service.

```python
from QCblog import db, login_manager
from flask_login import UserMixin
from datetime import datetime

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(20), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    image_file = db.Column(db.String(20), nullable=False, default='default.jpg')
    password = db.Column(db.String(60), nullable=False)
    posts = db.relationship('Post', backref='author', lazy=True)
    recipes = db.relationship('Recipe', backref='author', lazy=True)

    def __repr__(self):
        return '<User {}>'.format(self.username, self.email, self.image_file)


class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    date_posted = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    content = db.Column(db.Text, nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

    def __repr__(self):
        return '<Post {}>'.format(self.title, self.date_posted)


class Recipe(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    recipe = db.Column(db.Text, nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

    def __repr__(self):
        return '<Recipe {}>'.format(self.title, self.date_posted)
```

Flask features such as flask_login, render_template, flash, redirect etc. were also utilised to add more functionality to the web service. The flask_login functionality allows for the service to carry out certain functions such as using the information of the current user, ensuring that a page can only be accessed if the user is logged in and also allows for users to logout. The render feature allows for html templets to be rendered. Flash is used to display message to the user.

Redirect is used to move the webpage from one page to another using the app route. Bcrypt was also utilised during development to allow the hashing of passwords.

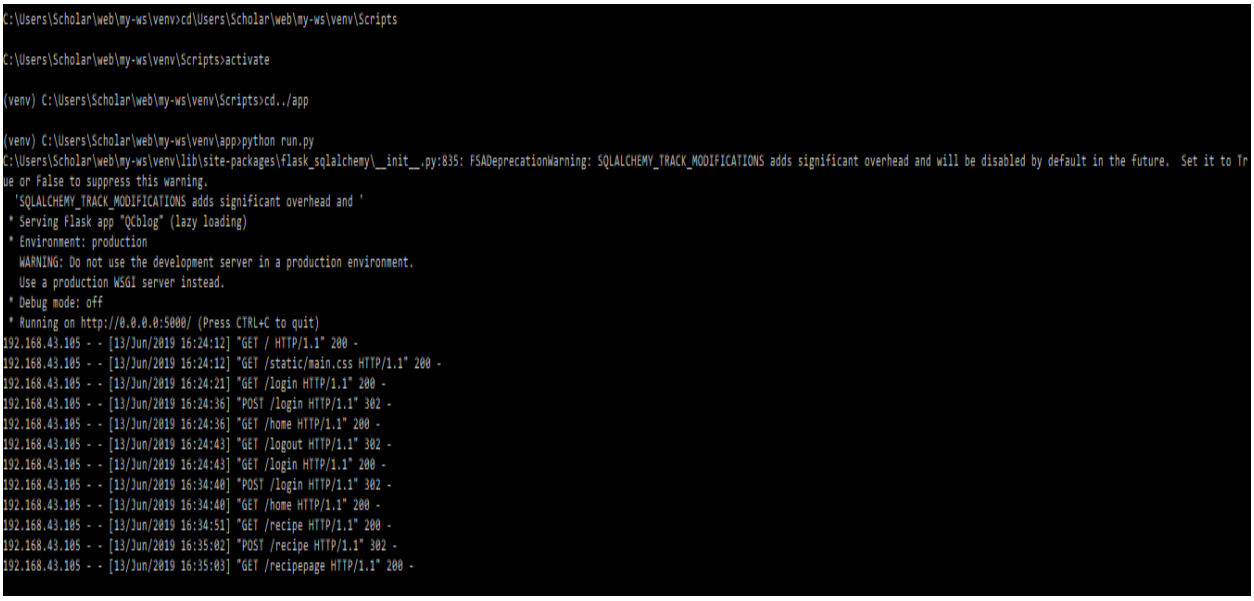
```
from flask import render_template, flash, redirect, url_for, flash, request
from QCblog.forms import RegistrationForm, LoginForm, RecipeForm, PostForm
from QCblog import app, db, bcrypt
from QCblog.models import User, Post, Recipe
from flask_login import login_user, current_user, logout_user, login_required



@app.route("/")
@app.route("/home")
def home():
    posts=Post.query.all()
    return render_template('home.html', posts=posts)

@app.route("/recipepage")
def recipePage():
    recipes=Recipe.query.all()
    return render_template('recipepage.html', recipes=recipes)


ml="login has an error refer to video3"
@app.route("/register", methods=['GET', 'POST'])
def register():
    if current_user.is_authenticated:
        return redirect(url_for('home'))
    form = RegistrationForm()
    if form.validate_on_submit():
        H_password = bcrypt.generate_password_hash(form.password.data).decode('utf-8')
        user = User(username=form.username.data, email=form.email.data, password=H_password)
        db.session.add(user)
        db.session.commit()
        flash('Your account has been created!', 'success')
        return redirect(url_for('login'))
    return render_template('register.html', title="Register", form=form)
```

The development of the website was done using HTML, CSS and Bootstrap. HTML (hypertext markup language) was used to create the content for website which was then styled using CSS and Bootstrap templates.

```
{%extends "layout.html"%}
{%block content%}
    <div class="content-section">
        <form method="POST" action="">
            {{form.hidden_tag()}}
            <fieldset class="form-group">
                <legend class="border-bottom mb-4">Join Today</legend>
                <div class="form-group">
                    {{form.username.label(class="form-control-label")}}

                    {% if form.username.errors %}
                        {{ form.username(class="form-control form-control-lg is-invalid") }}
                        <div class="invalid-feedback">
                            {% for error in form.username.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{form.username(class="form-control form-control-lg")}}
                    {% endif %}
                </div>
                <div class="form-group">
                    {{form.email.label(class="form-control-label")}}

                    {% if form.email.errors %}
                        {{ form.email(class="form-control form-control-lg is-invalid") }}
                        <div class="invalid-feedback">
                            {% for error in form.email.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{form.email(class="form-control form-control-lg")}}
                    {% endif %}
                </div>
                <div class="form-group">
                    {{form.password.label(class="form-control-label")}}

                    {% if form.password.errors %}
                        {{ form.password(class="form-control form-control-lg is-invalid") }}
```

```
<!DOCTYPE html>
<html>
<head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT
    <link rel="stylesheet" type="text/css" href="{{url_for('static',filename='main.css')}}">


    {%if title %}
        <title>Quick Cart Blog - {{title}}</title>
    {%else%}
        <title>Quick Cart Blog</title>
    {%endif%}
    <link rel="shortcut icon" href="C:\Users\Scholar\web\my-ws\venv\app\QCblog\templates\QC.png" type="image/png"/>
</head>
<body background = supermarket.jpg>
    <header class="site-header">
        <nav class="navbar navbar-expand-md navbar-dark bg-steel fixed-top">
        <div class="container">
          <a class="navbar-brand mr-4" href="/">Quick Cart Blog</a>
          <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarToggle" aria-controls="navbarToggle" aria-expanded="false" aria-label="Toggle navigat
            <span class="navbar-toggler-icon"></span>
          </button>
          <div class="collapse navbar-collapse" id="navbarToggle">
            <div class="navbar-nav mr-auto">
              <a class="nav-item nav-link" href="{{url_for('home')}}">Home</a>
              <a class="nav-item nav-link" href="{{url_for('recipePage')}}">Recipes</a>
            </div>
            <!-- Navbar Right Side -->
            <div class="navbar-nav">
            {% if current_user.username == "Quick Cart" %}
                <a class="nav-item nav-link" href="{{url_for('post')}}">Create Post</a>
            {% endif %}
            {% if current_user.is_authenticated %}
                <a class="nav-item nav-link" href="{{url_for('recipe')}}">Add Recipe</a>
```

After the completion of the coding aspect of the project we then launched the application using the command line thus launching the server.



As seen in the above picture, when the site is launched it utilises GET and POST methods in order to retrieve data from the database and handle data being submitted by the user.