Developing A Stock Market Web App Using Python Flask

What is Flask Python?

Flask is a web framework, it's a Python module that lets you develop web applications easily. it's a microframework that doesn't include an ORM (Object Relational Manager) or such features.

It does have many cool features like url routing, template engine. It is a WSGI web app framework.

What is web framework?

A Web Application Framework or a simply a Web Framework represents a collection of libraries and modules that enable web application developers to write applications without worrying about low-level details such as protocol, thread management, and so on.

What is Flask?

Flask is a web application framework written in Python. It was developed by Armin Ronacher.
Flask is based on the Werkzeug WSGI toolkit and the Jinja2 template.

Flask is often referred to as a microframework. It is designed to keep the core of the application simple and scalable.

Instead of an abstraction layer for database support, Flask supports extensions to add such capabilities to the application.

WSGI:

The Web Server Gateway Interface (Web Server Gateway Interface, WSGI) has been used as a standard for Python web application development. WSGI is the specification of a common interface between web servers and web applications.

Werkzeug:

Werkzeug is a WSGI toolkit that implements requests, response objects, and utility functions. This enables a web frame to be built on it. The Flask framework uses Werkzeug as one of its bases.

Jinja2:

jinja2 is a popular template engine for Python. A web template system combines a template with a specific data source to render a dynamic web page.

This allows you to pass Python variables into HTML templates

Lets create a stock market web app!

Steps in Creating a Stock Market Web app:

Step1: Installing Flask

Step2: Creating a base app

Step3: Using HTML templates

Step4: Setting up database for stocks and users

Step5: Displaying the stock items

Step6:Creating Registration form and Login form

Step7:validating Registration form and Login form

Step1: Installing Flask:

In this step, you'll activate your Python environment and install Flask using the pip package installer.

Syntax: pip install flask

Step2: Creating a base application:

In this step, you'll make a small web application inside a Python file and run it to start the server, which will display some information on the browser.

Code:

```
# Importing flask module in the project is mandatory An object of Flask class is our WSGI application.#

#from flask import Flask  Flask constructor takes the name of current module (__name__) as argument.#

app = Flask(__name__)

    # The route() function of the Flask class is a decorator,

    # which tells the application which URL should call the associated function.

@app.route('/')

      # '/' URL is bound with hello_world() function.

def hello_world():

      return 'Hello World'

# main driver function

if __name__ == '__main__':

    # run() method of Flask class runs the application on the local development server.

      app.run()
```

Output

```
 * Serving Flask app "hello" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 813-894-335
```

The application is running locally on the URL http://127.0.0.1:5000/, 127.0.0.1 is the IP that represents your machine's localhost and :5000 is the port number.

Open a browser and type in the URL http://127.0.0.1:5000/, you will receive the string Hello, World! as a response, this confirms that your application is successfully running.

Step3:Using of HTML Templates:

Currently your application only displays a simple message without any HTML. Web applications mainly use HTML to display information for the visitor.

Flask provides a render_template() helper function that allows use of the Jinja template engine. This will make managing HTML much easier by writing your HTML code in .html files as well as using logic in your HTML code. You'll use these HTML files, (*templates*) to build all of your application pages, such as the main page where you'll display the current blog posts, the page of the stocks, the page where the user can add a new stocks, and so on.

Code:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')

def index():

    return render_template('index.html')
```

- The index() view function returns the result of calling render_template() with index.html as an argument, this tells render_template() to look for a file called index.html in the *templates folder*. Both the folder and the file do not yet exist, you will get an error if you were to run the application at this point.
- So before run the application you should create templates folder and HTML file.

- In addition to the templates folder, Flask web applications also typically have a static folder for hosting static files, such as CSS files, JavaScript files, and images the application uses.

- You can create a style.css style sheet file to add CSS to your application. First, create a directory called static inside your main directory.

```
{% extends 'base.html' %}
{% block title %}
    Home Page
{% endblock %}

{% block content %}
    This is our content for the Home Page
{% endblock %}

```

However, the following highlighted parts are specific to the Jinja template engine:

- {% block title %} {% endblock %}: A block that serves as a placeholder for a title, you'll later use it in other templates to give a custom title for each page in your application without rewriting the entire <head> section each time.
- {{ url_for('index')}}: A function call that will return the URL for the index() view function. This is different from the past url_for() call you used to link a static CSS file, because it only takes one argument, which is the view function's name, and links to the route associated with the function instead of a static file.
- {% block content %} {% endblock %}: Another block that will be replaced by content depending on the *child template* (templates that inherit from base.html) that will override it.
- Using Template inheritance you will avoid typing the same code again and again.

Step4:Creating data base for User and Stocks:

Using raw SQL in the Flask Web application to perform CRUD operations on the database can be cumbersome.

Instead, SQLAlchemy, the Python Toolkit is a powerful OR Mapper, which provides application developers with the full functionality and flexibility of SQL.

Flask-SQLAlchemy is a Flask extension that adds support for SQLAlchemy to the Flask application.

Step1:Insatlling Sql Alchemy:

```
pip install flask-sqlalchemy
```

Step2: You need to import the SQLAlchemy class from this module:

```
from flask_sqlalchemy import SQLAlchemy
```

Step3:Now create a Flask application object and set the URI for the database to use:

```
app = Flask (__name__)
app.config ['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///students.sqlite3'
```

Step 4 - then use the application object as a parameter to create an object of class SQLAlchemy.The object contains an auxiliary function for the ORM operation.It also provides a parent Model class that uses it to declare a user-defined model.

```
db = SQLAlchemy(app)
```

Step 5 - To create/use the database mentioned in the URI, run the create_all() method.

```
db.create_all()
```

Step6:Add data to the models:

```
db.session.add(User)/db.session.add(Item)

db.session.commit()
```

In the code snippet below User & Item model is Created.

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer(), primary_key=True)
    username = db.Column(db.String(length=30), nullable=False, unique=True)
    email = db.Column(db.String(length=50), nullable=False, unique=True)
    password= db.Column(db.String(length=150), nullable=False)
    budget = db.Column(db.Integer(), nullable=False, default=10000)
    items = db.relationship('Item', backref='owned_user', lazy=True)

    def __repr__(self):
        return f"User('{self.username}','{self.email}')"


class Item(db.Model):
    id = db.Column(db.Integer(), primary_key=True)
    name = db.Column(db.String(length=30), nullable=False, unique=True)
    price = db.Column(db.Integer(), nullable=False)
    barcode = db.Column(db.String(length=12), nullable=False, unique=True)
    description = db.Column(db.String(length=1024), nullable=False,
unique=True)
    owner = db.Column(db.Integer(), db.ForeignKey('user.id'))
    def __repr__(self):
        return
f"Item('{self.name}','{self.price}','{self.barcode}','{self.description}','{se
lf.owner}')"
```

Step7:Database Relationships.

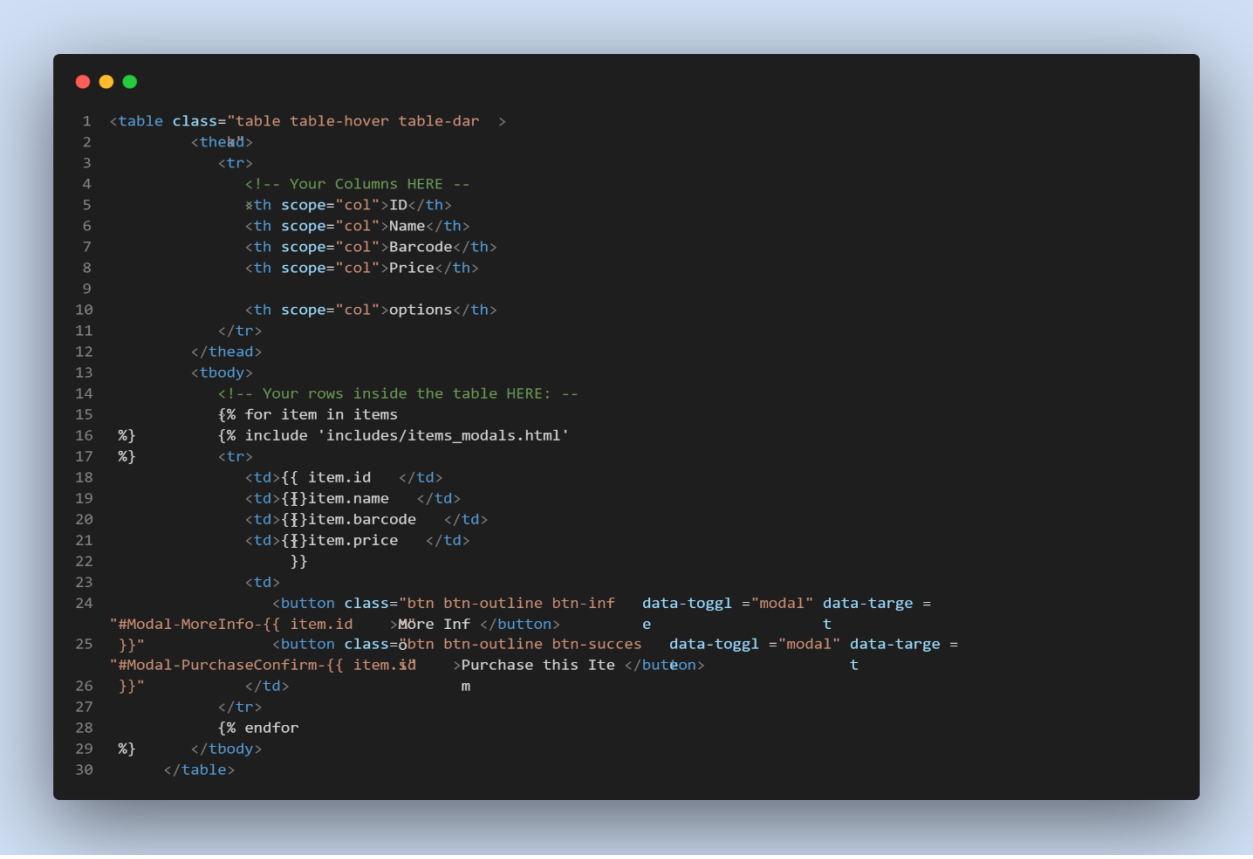
SQLAlchemy connects to relational databases and what relational databases are really good at are relations. As such, we shall have an example of an application that uses two tables that have a relationship to each other:

Step5:Displaying stock items:

```
@app.route('/stocks')
def Stocks():
    items = Item.query.all()
    return render_template(stocks.html", items=items)
```

by sending the data from the database to html template using jinja template you can view the stock items at the web page.

Example:



```
<table class="table table-hover table-dar  >
        <thead>
           <tr>
              <!-- Your Columns HERE --
              <th scope="col">ID</th>
              <th scope="col">Name</th>
              <th scope="col">Barcode</th>
              <th scope="col">Price</th>

              <th scope="col">options</th>
           </tr>
        </thead>
        <tbody>
           <!-- Your rows inside the table HERE: --
           {% for item in items
 %}        {% include 'includes/items_modals.html'
 %}        <tr>
              <td>{{ item.id   </td>
              <td>{}item.name   </td>
              <td>{}item.barcode   </td>
              <td>{}item.price   </td>
                  }}
              <td>
                 <button class="btn btn-outline btn-inf   data-toggl ="modal" data-targe =
"#Modal-MoreInfo-{{ item.id    >More Inf </button>         e                  t
 }}"             <button class=btn btn-outline btn-succes   data-toggl ="modal" data-targe =
"#Modal-PurchaseConfirm-{{ item.id    >Purchase this Ite </buteon>              t
 }}"          </td>                   m
           </tr>
           {% endfor
 %}     </tbody>
      </table>
```

Step6:creating registration and validation form:

Flask forms:

The form is the basic element that lets users interact with our web application.

Flask alone doesn't do anything to help us handle forms, but the Flask-WTF extension lets us use the popular WTForms package in our Flask applications.

This package makes defining forms and handling submissions easy.

Code:

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField


class RegisterForm(FlaskForm):
    username = StringField(label='User Name:')
    email_address = StringField(label='Email Address:')
    password1 = PasswordField(label='Password:')
    password2 = PasswordField(label='Confirm Password:')
    submit = SubmitField(label='Create Account')
```

```
class loginForm(FlaskForm):
    username = StringField(label='User Name:')
    password1 = PasswordField(label='Password:')
    submit = SubmitField(label='Create Account')
```

after creating the forms you need to create routes /url for the pages.

```
@app.route('/register')
def register_page():
    form = RegisterForm()
    return render_template('register.html', form=form)

@app.route('/login')
def login_page():
    form = login()
    return render_template('register.html', form=form)
```

using jinja template form content will show in web pages using Html templates

example:

```
{% block content %}
<body class="text-center">
<div class="container">
    <form method="POST" action="">

        {{ form.hidden_tag() }}

        <h1 class="h3 mb-3 font-weight-normal">
         Please Create your Account
     </h1>
     <br>
     {{ form.username.label() }}
     {{ form.username(class="form-control", placeholder="User Name") }}

     {{ form.email.label() }}
     {{ form.email(class="form-control", placeholder="Email Address") }}

     {{ form.password.label() }}
     {{ form.password(class="form-control", placeholder="Password") }}

     {{ form.confirm_password.label() }}
     {{ form.confirm_password(class="form-control", placeholder="Confirm
Password") }}

     <br>
```

```
 {{ form.submit(class="btn btn-lg btn-block btn-primary") }}
 </form>
 <a href="{{url_for('login')}}">already have an account ? Login here></a>
 </div>
</body>
{% endblock %}
```

Step:7:validating the registration and login forms:

now the pages will display the forms ,while entering the datas there may be unique constraint error /or some other errors may occur for that the form needs to be validated for errors before they occurring.

code:

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField,BooleanField,
SubmitField, HiddenField
from wtforms.validators import Length, EqualTo, Email, DataRequired,
ValidationError
from app1 import User


class RegisterForm(FlaskForm):
    def validate_username(self, username_to_check):
        user = User.query.filter_by(username=username_to_check.data).first()
        if user:
            raise ValidationError('Username already exists! Please try a
different username')

    def validate_email_address(self, email_address_to_check):
        email_address =
User.query.filter_by(email_address=email_address_to_check.data).first()
        if email_address:
            raise ValidationError('Email Address already exists! Please try a
different email address')

    username = StringField(label='User Name:', validators=[Length(min=2,
max=30), DataRequired()])
    email = StringField(label='Email Address:', validators=[Email(),
DataRequired()])
    password = PasswordField(label='Password:', validators=[Length(min=6),
DataRequired()])
    confirm_password = PasswordField(label='Confirm Password:',
validators=[EqualTo('password'), DataRequired()])
    submit = SubmitField(label='Create Account')
```

```
class LoginForm(FlaskForm):
    username = StringField(label='User Name:', validators=[DataRequired()])
    password = PasswordField(label='Password:', validators=[DataRequired()])
    remember=BooleanField("Remember me")
    submit = SubmitField(label='Sign in')
```

this validators will rectify the errors of creating unique constaint errors before it occurs

if you hit submit the webpage will show method not allowed , for that we need to validate the submit button.

1.If they hit submit of the register form it should create a new User

1.Code:

```
@app.route("/register",methods=['GET','POST'])
def register_page():
    from form import RegisterForm
    form=RegisterForm()
    if form.validate_on_submit():
        user = User(username=form.username.data,
                                email=form.email.data,

                                password=form.password.data)
        db.session.add(user)
        db.session.commit()
        flash("your account has been created ")
        return redirect(url_for('login'))
    if form.errors != {}: #If there are not errors from the validations
        for err_msg in form.errors.values():
            flash(f'There was an error with creating a user: {err_msg}',
category='danger')

    return render_template('register.html', form=form)
```

now a new user will be created and user details will be stored in the database.after creating the user you can login using login page

2.for validating login:

Code:

```
from flask_login import LoginManager
from flask_login import login_user,current_user
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

```
@app.route("/login",methods=['GET','POST'])
def login():


    from form import LoginForm
    form=LoginForm()
    if form.validate_on_submit():
       user=User.query.filter_by(username=form.username.data).first()
       if form.username.data==user.username and
form.password.data==user.password:
            login_user(user,remember=True)
            flash("login successful","success")
            return redirect(url_for('stocks_page'))

    else:
        flash("login unsuccesful please check username and password","danger")
    return render_template("login.html",form=form)
```

Flash:

The flash() method is **used to generate informative messages in the flask**. It creates a message in one view and renders it to a template view function called next. In other words, the flash() method of the flask module passes the message to the next request which is an HTML template.

Messages in html:

```
{% block content %}

<title>My Application</title>
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
      <li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
{% block body %}{% endblock %}
```

Logout:

You can simply logout by using logout route ,

```
@app.route("/logout")
def logout():
    logout_user()
    return redirect(url_for("home"))
```