Project Title

"Responsive Blog Website with Admin Control and Content Management."

Tech Stack Summary

Flask, Bootstrap, SQLAlchemy, PostgreSQL, Git, Oracle Cloud.

Debojyoti Chattoraj

Introduction

This project is a fully functional blog website built using Flask as the backend framework and Bootstrap for the frontend design. The website allows users to create, edit, and delete blog posts, register and log in, and interact with other users through a comment system.

This blog website showcases a full-stack implementation that reflects my end-to-end development capabilities. Using **Flask** for the backend, I created dynamic routes and integrated form logic for user-controlled content management. On the data layer, I configured and optimized a **PostgreSQL** database, interfacing through **SQLAlchemy** to ensure secure and scalable model operations. The frontend, built with **Bootstrap**, maintains responsiveness and user-friendly navigation across devices. Together, these components demonstrate my proficiency in designing structured, maintainable, and aesthetically pleasing web applications that blend performance with usability.

Objective

Create a user-friendly blog website.

Enable admin control for managing tools and certificates.

Ensure scalability and responsive design for multiple device.

Purpose

Describe the blog’s core intent (e.g., personal knowledge hub, ML project showcase, tool explanation, etc.)

Tech Stack Overview

Layer	Tools Used	Why Chosen
Backend	Flask, SQLAlchemy, Flask-WTF, Flask-Login	Lightweight, scalable, easy to integrate.
Frontend	HTML, CSS, Bootstrap	Responsive design and rapid prototyping.
Database	PostgreSQL	Reliable relational data storage
Deployment	Linux Server on Oracle Cloud, Gunicorn (Web server)	99.9 % up time at free tier, easy deployment.
Messaging Integration	Twilio WhatsApp API	Easy Integration
Hosting Support	Caddy for reverse proxy setup	Inbuild Certification

Feature & Functionality

1. **User Management**: To implement user registration, login, and logout functionalities in my Flask application, I used the Flask-Login extension, which provides user session management capabilities. I have used werkzeug.security from the Werkzeug library to securely store User Passwords as hashed password and authentication. I have introduced WTForms for user registration and login.
2. **Role-Based Access Control**: I have introduced role-based privilege control to manage user activity on the website. For example, a user with 'admin' privilege only can write a post or manage user privilege (can grant other users admin or user privilege). I have extended this role-based access control to the extent that users only can edit the blog that they have created.
3. **Dynamic Rendering**: Jinja2 templating for routing and content loading.
4. **Admin Dashboard**: Create, edit, and delete blog posts. Manage blog categories for better organization. Save blog as draft for later editing. Preview unpublished posts and publish when ready.

System Architecture

1. **Presentation Layer (Frontend):**

The user interacts with the application through dynamically rendered HTML pages, styled using **Bootstrap 5** and enriched with **CKEditor** for content editing.

Include Bootstrap JS & CSS: Customized Bootstrap by overriding its styles with my own custom CSS. This allows to maintain the responsive features of Bootstrap while giving the site a unique look. Visit this header.html file for code reference. [Ref. Fig. 1]

I included the Bootstrap JavaScript at the end of the HTML file to enable Bootstrap components. I enclosed these codes in the footer.html file. Visit the footer.html file for code reference.

Additional Pages: Created additional HTML files for different webpages of the site, such as 'home.html', 'about.html', and 'contact.html', etc. Visit templates for reference.

Rendering Forms: I used Jinja templating to render forms and include header and footer in the HTML templates. For instance, a login form template (login.html). [Ref. Fig. 2]

Building Forms: I used WTForms and Object Oriented Programming techniques to build different forms required for different activities. For instance, a login form is created using WTForms, as below. [Ref. Fig. 3]

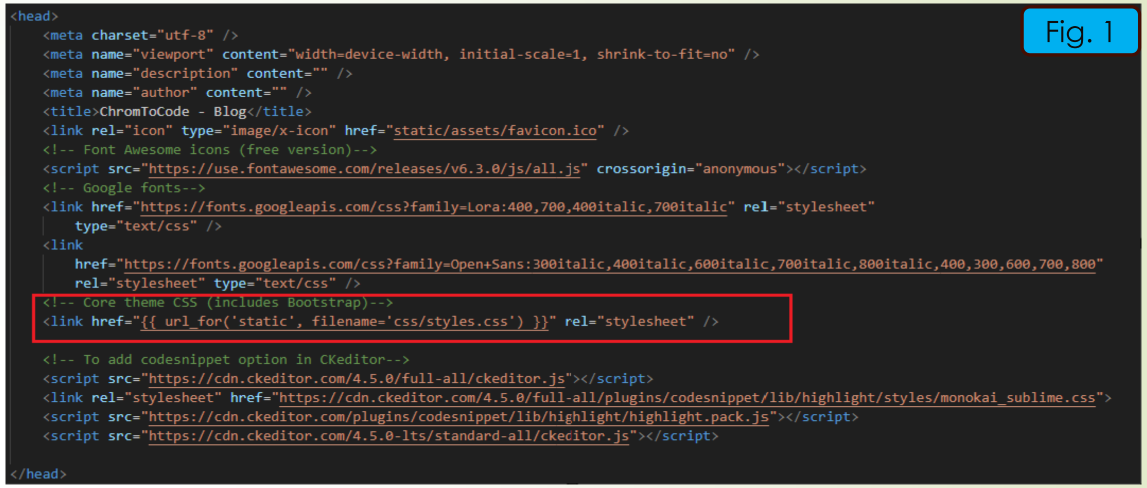


Fig. 1

```html
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
    <meta name="description" content="" />
    <meta name="author" content="" />
    <title>ChromToCode - Blog</title>
    <link rel="icon" type="image/x-icon" href="static/assets/favicon.ico" />
    <!-- Font Awesome icons (free version)-->
    <script src="https://use.fontawesome.com/releases/v6.3.0/js/all.js" crossorigin="anonymous"></script>
    <!-- Google fonts-->
    <link href="https://fonts.googleapis.com/css?family=Lora:400,700,400italic,700italic" rel="stylesheet"
        type="text/css" />
    <link
        href="https://fonts.googleapis.com/css?family=Open+Sans:300italic,400italic,600italic,700italic,800italic,400,300,600,700,800"
        rel="stylesheet" type="text/css" />
    <!-- Core theme CSS (includes Bootstrap)-->
    <link href="{{ url_for('static', filename='css/styles.css') }}" rel="stylesheet" />

    <!-- To add codesnippet option in CKeditor-->
    <script src="https://cdn.ckeditor.com/4.5.0/full-all/ckeditor.js"></script>
    <link rel="stylesheet" href="https://cdn.ckeditor.com/4.5.0/full-all/plugins/codesnippet/lib/highlight/styles/monokai_sublime.css">
    <script src="https://cdn.ckeditor.com/plugins/codesnippet/lib/highlight/highlight.pack.js"></script>
    <script src="https://cdn.ckeditor.com/4.5.0-lts/standard-all/ckeditor.js"></script>

</head>
```

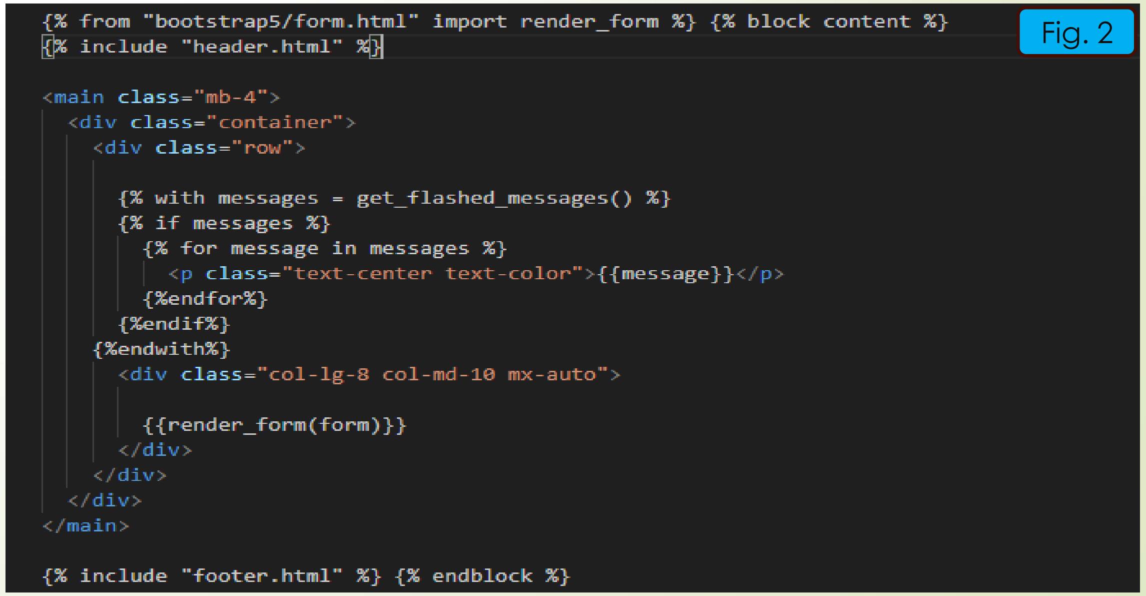


Fig. 2

```html
{% from "bootstrap5/form.html" import render_form %} {% block content %}
{% include "header.html" %}

<main class="mb-4">
  <div class="container">
    <div class="row">

      {% with messages = get_flashed_messages() %}
      {% if messages %}
        {% for message in messages %}
          <p class="text-center text-color">{{message}}</p>
        {%endfor%}
      {%endif%}
    {%endwith%}
      <div class="col-lg-8 col-md-10 mx-auto">

        {{render_form(form)}}
      </div>
    </div>
  </div>
</main>

{% include "footer.html" %} {% endblock %}
```

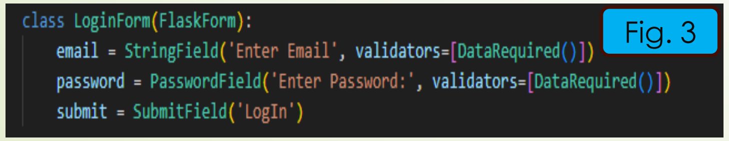


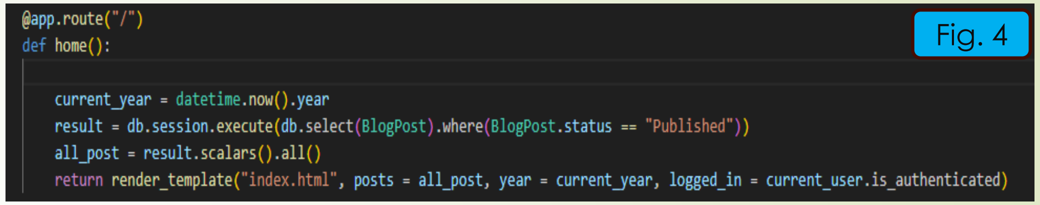
Fig. 3

```python
class LoginForm(FlaskForm):
    email = StringField('Enter Email', validators=[DataRequired()])
    password = PasswordField('Enter Password:', validators=[DataRequired()])
    submit = SubmitField('LogIn')
```

2. Application Layer (Flask Backend):

This layer handles all core logic, routing, session management, user authentication, and admin access control.

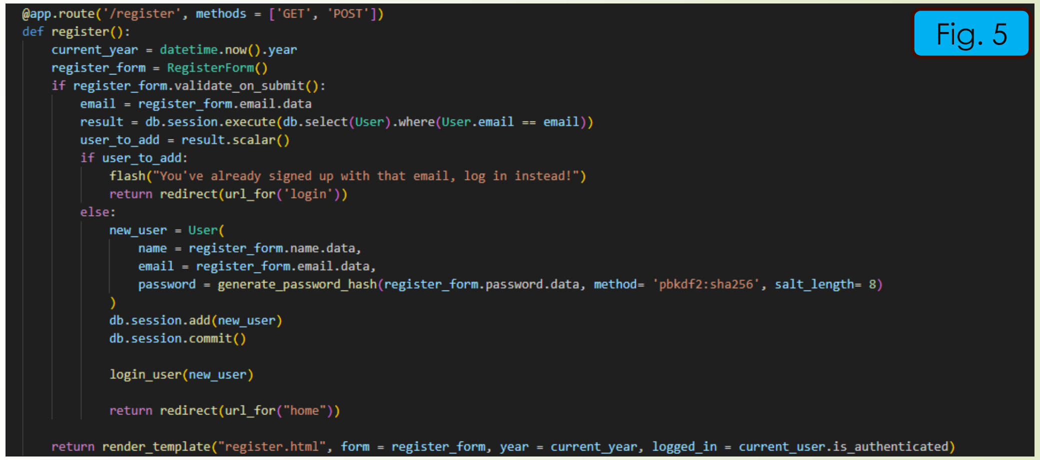
Flask Routes: Flask routes are created to render all the templates when accessed. For example, this home route displays the homepage (index.html) with a list of blog posts fetched from the database, which is supplied as input in the 'index.html' page. Visit main.py to refer to all the routes created and their functions. [Ref. Fig. 4]



```python
@app.route("/")
def home():

    current_year = datetime.now().year
    result = db.session.execute(db.select(BlogPost).where(BlogPost.status == "Published"))
    all_post = result.scalars().all()
    return render_template("index.html", posts = all_post, year = current_year, logged_in = current_user.is_authenticated)
```

Fig. 4

Handling Forms: I have used Flask-WTF, an extension of the Flask framework that integrates with WTForms, to handle form submissions in Flask. I have already discussed the Form building and rendering in the Frontend Development section, hence, here I will discuss Form Validation. Here is the approach I followed to validate and handle the registration form in the register route. [Ref. Fig. 5]

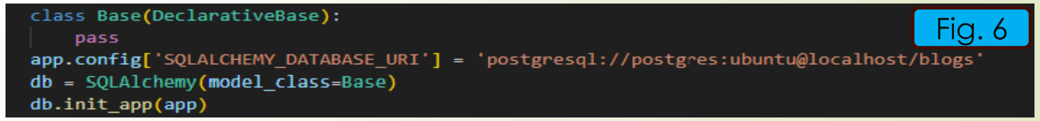


```python
@app.route('/register', methods = ['GET', 'POST'])
def register():
    current_year = datetime.now().year
    register_form = RegisterForm()
    if register_form.validate_on_submit():
        email = register_form.email.data
        result = db.session.execute(db.select(User).where(User.email == email))
        user_to_add = result.scalar()
        if user_to_add:
            flash("You've already signed up with that email, log in instead!")
            return redirect(url_for('login'))
        else:
            new_user = User(
                name = register_form.name.data,
                email = register_form.email.data,
                password = generate_password_hash(register_form.password.data, method= 'pbkdf2:sha256', salt_length= 8)
            )
            db.session.add(new_user)
            db.session.commit()

            login_user(new_user)

            return redirect(url_for("home"))

    return render_template("register.html", form = register_form, year = current_year, logged_in = current_user.is_authenticated)
```

Fig. 5

Database Integration: For this blog website, I used the **PostgreSQL** relational database and **Flask-SQLAlchemy** which is a popular ORM (Object-Relational Mapping) tool that makes it easy to work with databases in Flask applications.

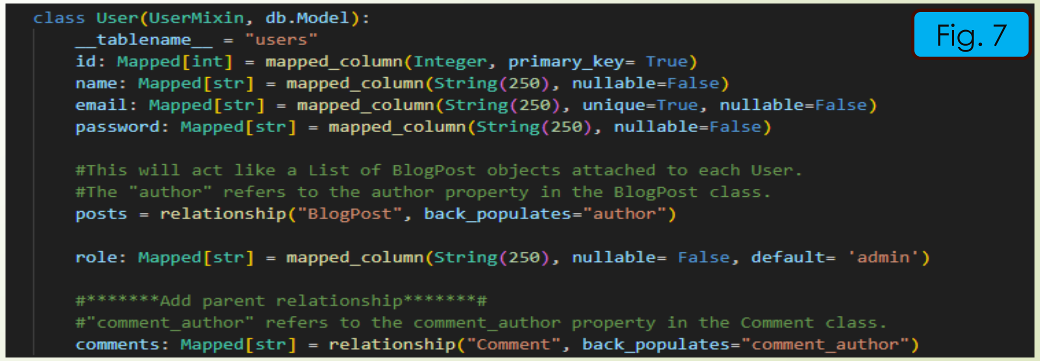
Configure the Database: In the Flask application's configuration, I specified the database URI. [Ref. Fig. 6]



```python
class Base(DeclarativeBase):
    pass
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://postgres:ubuntu@localhost/blogs'
db = SQLAlchemy(model_class=Base)
db.init_app(app)
```

Fig. 6

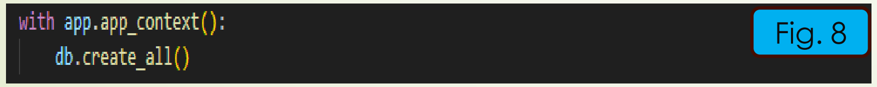
Creating Database Models: I created models to represent your database tables. For example, I have created a User model to define the 'users' table in the 'blogs' database. [Ref. Fig. 7]



```python
class User(UserMixin, db.Model):
    __tablename__ = "users"
    id: Mapped[int] = mapped_column(Integer, primary_key= True)
    name: Mapped[str] = mapped_column(String(250), nullable=False)
    email: Mapped[str] = mapped_column(String(250), unique=True, nullable=False)
    password: Mapped[str] = mapped_column(String(250), nullable=False)

    #This will act like a List of BlogPost objects attached to each User.
    #The "author" refers to the author property in the BlogPost class.
    posts = relationship("BlogPost", back_populates="author")

    role: Mapped[str] = mapped_column(String(250), nullable= False, default= 'admin')

    #*******Add parent relationship*******#
    #"comment_author" refers to the comment_author property in the Comment class.
    comments: Mapped[str] = relationship("Comment", back_populates="comment_author")
```

Fig. 7

Initializing the database and Creating tables: After defining the models, I initialized the database and created the tables. [Ref. Fig. 8]



```python
with app.app_context():
    db.create_all()
```

Fig. 8

3. **Content Lifecycle Flow:**

- **User Registration/Login:**

New users register with hashed passwords. Authenticated users get role-based access.

- **Post Creation & Editing:**

Admins can create posts using a form that supports draft and publish options.

- **Post Categorization & Filtering:**

Posts are filtered by subject (e.g., "Code", "Chemistry") and status ("Published", "Draft").

- **Commenting & Interaction:**

Authenticated users can comment on posts. Each comment is linked to its author and post.

4. **Deployment:**

For the deployment of this site, I have used a Linux (Ubuntu) Virtual Private Server (VPS) on Oracle Cloud. I have used Reverse-proxied using **Caddy** for domain routing and HTTPS setup and Hosted via **Gunicorn** WSGI server to run the Flask Server on the VPS. To read more about this deployment process please visit this link. Finally, I mapped the IP address of the server to a domain.

5. **Summary:**

This modular architecture ensures clean separation of concerns, security, and maintainability. The use of decorators, form handling, ORM, and third-party APIs showcases a complete full-stack system built with modern Flask practices.

Visit the fully functional Blog Website here: https://chromtocode.in/

GitHub Repository: https://github.com/CwDebojyoti/My_Blog.git

--End of Document--