

# Day 61: Building Advanced Forms with Flask-WTF

Today's focus is on moving beyond basic HTML forms to create secure, validated, and easily manageable web forms using the **Flask-WTF** extension.

## Key Concepts

- **CSRF Protection:** Cross-Site Request Forgery is an attack where a malicious site tricks a user into performing an unwanted action on a trusted site. Flask-WTF provides automatic CSRF protection.
- **Flask-WTF & WTForms:** Flask-WTF is a Flask extension that integrates the WTForms library, allowing you to define your forms as Python classes.
- **Form Validation:** Ensuring user input meets specific criteria (e.g., a field is not empty, an email is in the correct format).
- **Rendering Forms:** Injecting the form fields into your HTML templates cleanly.

## Creating a Form Class

You define a form by creating a class that inherits from `FlaskForm`. Each field in the form is an object from `wtforms` (e.g., `StringField`, `PasswordField`, `SubmitField`).

**Validators** are arguments passed to a field to enforce rules.

```
# forms.py
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email, Length

class LoginForm(FlaskForm):
    # DataRequired validator ensures the field is not submitted empty.
    email = StringField(label='Email', validators=[DataRequired(), Email()])

    # We can chain multiple validators.
    password = PasswordField(label='Password', validators=[DataRequired(), Length(min=8)])

    submit = SubmitField(label="Log In")
```

# Handling Forms in Your Flask App

1. **Secret Key:** Flask-WTF requires a `SECRET_KEY` in your app configuration for CSRF protection.
2. **Instantiation:** Create an instance of your form class in your route.
3. **Validation:** Use `form.validate_on_submit()` to check if the form was submitted (POST request) and if all validators passed.
4. **Data Access:** If validation is successful, access the data with `form.field_name.data`.

```
# main.py
from flask import Flask, render_template
from forms import LoginForm

app = Flask(__name__)
# This key should be a long, random, secret string in a real app.
app.config['SECRET_KEY'] = 'a-super-secret-key'

@app.route("/login", methods=["GET", "POST"])
def login():
    login_form = LoginForm()
    # This block runs only on a POST request when all validators pass.
    if login_form.validate_on_submit():
        # Access the validated data
        email = login_form.email.data
        password = login_form.password.data
        if email == "admin@email.com" and password == "12345678":
            return render_template("success.html")
        else:
            return render_template("denied.html")
    # This runs on a GET request or if validation fails.
    return render_template('login.html', form=login_form)
```

## Rendering the Form in HTML

In your Jinja2 template, you can render the form fields easily.

- `form.hidden_tag()` : This is crucial! It adds the hidden CSRF token field.
- `form.<field_name>.label` : Renders the `<label>` tag.
- `form.<field_name>()` : Renders the input field itself (e.g., `<input type="text">` ).
- `form.<field_name>.errors` : A list of validation errors for a specific field.

```

<form method="POST" novalidate>
    {{ form.hidden_tag() }}
    <p>
        {{ form.email.label }}<br>
        {{ form.email(size=30) }}
        {% for err in form.email.errors %}
        <span style="color: red;">{{ err }}</span>
        {% endfor %}
    </p>
    <p>
        {{ form.password.label }}<br>
        {{ form.password(size=30) }}
        {% for err in form.password.errors %}
        <span style="color: red;">{{ err }}</span>
        {% endfor %}
    </p>
    <p>{{ form.submit() }}</p>
</form>

```

## Day's Project: A Simple Login Page

The project for the day is to create a simple website with a login page that uses a `LoginForm` created with Flask-WTF. The form should validate that the email and password fields are filled correctly and then check them against hardcoded values to grant or deny access.

## Day 62: Coffee & Wifi Project

This day integrates multiple technologies—Flask, WTForms, Bootstrap, and CSV data handling—to build a practical web application.

### Key Concepts

- **Flask-Bootstrap:** An extension that simplifies integrating the Bootstrap CSS framework into Flask apps, making styling much faster.
- **Data Persistence with CSV:** Using Python's built-in `csv` module to read from and append to a CSV file, which acts as a simple database.
- **Advanced WTForms Fields:** Using different field types like `SelectField` for dropdowns and `URLField` for URLs.

# Integrating Flask-Bootstrap

After `pip install flask-bootstrap`, you initialize it in your app.

```
from flask import Flask
from flask_bootstrap import Bootstrap

app = Flask(__name__)
app.config['SECRET_KEY'] = 'some-secret'
Bootstrap(app) # Initialize Bootstrap
```

Then, in your base template ( `base.html` ), you can inherit from Bootstrap's base template.

```
{% extends "bootstrap/base.html" %}

{% block title %}Coffee & Wifi{% endblock %}

{% block content %}
<div class="container">
    </div>
{% endblock %}
```

## The Project: "Coffee & Wifi" Website

The goal is to build a website that lists cafes and their amenities (like WiFi strength and power socket availability). Users can also add new cafes to the list.

### 1. Display Cafes ( `/cafes` ):

- Read the `cafe-data.csv` file using `csv.reader`.
- Store the data in a list of lists.
- Pass this list to the `cafes.html` template and use a Jinja2 `for` loop to render the data in an HTML table.

### 2. Add a New Cafe ( `/add` ):

- Create a `CafeForm` using Flask-WTF with fields for the cafe name, location (Google Maps URL), opening/closing times, and `SelectField`s for ratings.
- The `SelectField` requires a `choices` argument, which is a list of tuples (value, label).

```

class CafeForm(FlaskForm):
    cafe = StringField('Cafe name', validators=[DataRequired()])
    location = URLField('Cafe Location on Google Maps (URL)', validators=[DataRequired(), U
# ... other fields
    wifi_rating = SelectField('Wifi Strength Rating', choices=["X", "👍", "👍👍", "👍👍👍
# ... more fields
    submit = SubmitField('Submit')

```

- When the form is submitted and validated, open the `cafe-data.csv` file in **append mode** (`'a'`).
- Create a new row with the data from `form.field.data` and write it to the CSV.
- Redirect the user back to the `/cafes` page.

```

# main.py
@app.route('/add', methods=["GET", "POST"])
def add_cafe():
    form = CafeForm()
    if form.validate_on_submit():
        # Create a new row as a string
        new_row = f"\n{form.cafe.data},{form.location.data},{...}"
        with open('cafe-data.csv', mode='a', encoding='utf-8') as csv_file:
            csv_file.write(new_row)
        return redirect(url_for('cafes'))
    return render_template('add.html', form=form)

```

## Day 63: Databases with SQLite & SQLAlchemy

This is a pivotal day where we move from fragile CSV files to robust relational databases.

### Key Concepts

- **Relational Databases:** Data is stored in tables with rows and columns. Relationships can be defined between tables. SQLite is a lightweight, file-based database engine perfect for development and small apps.
- **SQLAlchemy:** An Object Relational Mapper (ORM). It allows you to interact with your database using Python objects and methods instead of writing raw SQL queries. This makes your code more readable, portable, and less prone to SQL injection attacks.

- **CRUD Operations:** The four fundamental functions of database management: **C**reate, **R**ead, **U**ppdate, and **D**eleate.

## Setting up the Database with Flask-SQLAlchemy

1. Install: `pip install flask-sqlalchemy`
2. Configure and initialize in your app:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

##CREATE DATABASE
app.config['SQLALCHEMY_DATABASE_URI'] = "sqlite:///new-books-collection.db"
#Optional: But it will silence the deprecation warning in the console.
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

## Defining a Model (Table)

A model is a Python class that inherits from `db.Model`. Each attribute of the class represents a column in the table.

```
##CREATE TABLE
class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(250), unique=True, nullable=False)
    author = db.Column(db.String(250), nullable=False)
    rating = db.Column(db.Float, nullable=False)

# This line is needed to actually create the table in the database file.
# You only need to run this once.
db.create_all()
```

- `db.Column` : Defines a column.
- `db.Integer` , `db.String` , `db.Float` : Define the data type.
- `primary_key=True` : Marks this column as the unique identifier for each row.
- `unique=True` : Ensures all values in this column are unique.
- `nullable=False` : Makes this a required field.

# CRUD Operations with SQLAlchemy

```
# --- CREATE a new record ---
new_book = Book(title="Harry Potter", author="J. K. Rowling", rating=9.3)
db.session.add(new_book)
db.session.commit() # Don't forget to commit!

# --- READ all records ---
all_books = db.session.query(Book).all()

# --- READ a specific record by query ---
book = Book.query.filter_by(title="Harry Potter").first()

# --- READ a specific record by PRIMARY KEY ---
book_to_update = Book.query.get(1) # Gets the book with id=1

# --- UPDATE a record ---
book_to_update.title = "Harry Potter and the Chamber of Secrets"
db.session.commit()

# --- DELETE a record ---
book_to_delete = Book.query.get(1)
db.session.delete(book_to_delete)
db.session.commit()
```

## Day's Project: A Virtual Bookshelf

The project is to create a web application that manages a book collection.

- The homepage displays all books from the database in a list.
- There's an `add.html` page with a form to add new books.
- When the form is submitted, a new `Book` object is created and saved to the database.

## Day 64: My Top 10 Movies Website

This project builds on Day 63, applying SQLAlchemy CRUD operations in a more complex web application that interacts with an external API.

# Key Concepts

- **Applying CRUD in Flask:** Integrating Create, Read, Update, and Delete operations into Flask routes.
- **External API Integration:** Using the `requests` library to fetch data from an external API (like The Movie Database - TMDb).
- **Dynamic Routing for Updates:** Using routes like `/edit/<int:movie_id>` to target specific database records for modification.

## The Project: "Top 10 Movies" Website

The application allows a user to build a personal list of their top movies.

### 1. Database Model ( `Movie` ):

```
class Movie(db.Model):  
    id = db.Column(db.Integer, primary_key=True)  
    title = db.Column(db.String(250), unique=True, nullable=False)  
    year = db.Column(db.Integer, nullable=False)  
    description = db.Column(db.String(500), nullable=False)  
    rating = db.Column(db.Float, nullable=True) # User's rating  
    ranking = db.Column(db.Integer, nullable=True) # User's ranking  
    review = db.Column(db.String(250), nullable=True) # User's review  
    img_url = db.Column(db.String(250), nullable=False)
```

### 2. Main Page ( `/` ):

- **READ:** Fetches all movies from the DB:  
`all_movies = Movie.query.order_by(Movie.rating).all()` .
- **Logic for Ranking:** After fetching, loop through the sorted movies and assign a rank.
- Displays the movies on `index.html` .

### 3. Editing a Movie ( `/edit` ):

- This route handles both GET (to show the form) and POST (to update the data).
- It uses a dynamic URL: `@app.route("/edit", methods=["GET", "POST"])` .
- The movie's `id` is passed via a query parameter (e.g., `/edit?id=1` ).
- **UPDATE:**

```

@app.route("/edit", methods=["GET", "POST"])
def edit_movie():
    movie_id = request.args.get('id')
    movie_to_update = Movie.query.get(movie_id)
    if request.method == "POST":
        # Get data from the submitted form
        movie_to_update.rating = request.form["rating"]
        movie_to_update.review = request.form["review"]
        db.session.commit()
        return redirect(url_for('home'))
    return render_template("edit.html", movie=movie_to_update)

```

#### 4. Deleting a Movie ( /delete ):

- **DELETE:** Similar to edit, it gets the movie `id` from the URL.
- `movie_to_delete = Movie.query.get(movie_id)`
- `db.session.delete(movie_to_delete)`
- `db.session.commit()`

#### 5. Adding a Movie ( /add ):

- This is a two-step process.
- First, the user searches for a movie title on an `add.html` form.
- Your Flask app takes this query, calls the TMDb API, and gets a list of matching movies.
- It then renders a `select.html` page showing the results.
- When the user clicks a movie from the list, you get the TMDb movie ID. You then make a *second* API call to get detailed data for that specific movie.
- **CREATE:** You use this detailed data to create a new `Movie` object and add it to your own database.

## Day 65: Web Design School - Create a Website that People will Love

This day is a brief detour from backend coding to focus on the principles of **User Interface (UI)** and **User Experience (UX)** design. The goal is to make your websites not just functional, but also beautiful and intuitive.

# Key Concepts

- **Color Theory:** Understanding the color wheel, complementary colors, and triadic colors to create a visually appealing palette. Tools like [Coolors.co](https://coolors.co) are introduced for generating palettes.
- **Typography:** The art of arranging text to make it legible, readable, and appealing. Key ideas include choosing font pairings (one for headings, one for body text), font sizing, and line spacing. Google Fonts is the go-to resource.
- **UI Patterns:** Common, reusable solutions to design problems. Examples include navbars, cards, carousels, and forms. Frameworks like Bootstrap provide pre-built components for these patterns.
- **User Experience (UX):** Focuses on the overall feel of the application. Is it easy to use? Is the flow logical? Does it solve the user's problem without frustration?

## Day's Project: Redesign an Existing Project

There's no new coding project today. Instead, the task is to take one of the previous projects (like the Top 10 Movies website or the Coffee & Wifi site) and apply the design principles learned to improve its appearance and usability.

### Actionable Steps:

1. **Choose a Color Palette:** Go to [Coolors.co](https://coolors.co) and generate a palette of 3-5 colors. Apply these colors consistently throughout your site (e.g., primary color for buttons, background color, text color).
2. **Select Fonts:** Go to Google Fonts and pick two complementary fonts. Use one for all `<h1>`, `<h2>`, etc., and the other for all `<p>` tags and body text.
3. **Improve Layout:** Use CSS (or Bootstrap's grid system) to create a more organized and visually balanced layout. Instead of a simple list, maybe display movies or cafes as "cards".
4. **Add a Favicon:** Create a small icon for your website that appears in the browser tab.
5. **Refine with CSS:** Use CSS properties like `padding`, `margin`, `border-radius`, and `box-shadow` to give elements breathing room and a modern feel.

The goal is to transform a purely functional site into a professional-looking one.

## Day 66: Building Your Own API with RESTful Routing

Today, you switch from being a *consumer* of APIs to a *creator*. You will learn to build your own API that allows other programs to interact with your data.

# Key Concepts

- **API (Application Programming Interface):** A way for two or more computer programs to communicate with each other. A Web API allows communication over the internet, typically using HTTP.
- **REST (REpresentational State Transfer):** An architectural style for designing networked applications. It's not a strict protocol but a set of principles.
- **RESTful Routing:** A convention for structuring API endpoints based on the resources they manage and the HTTP verbs used to interact with them.

HTTP Verb	Endpoint	Action
GET	/cafes	<b>Read</b> - Get all cafes
GET	/cafes/<id>	<b>Read</b> - Get a specific cafe
POST	/cafes	<b>Create</b> - Add a new cafe
PUT / PATCH	/cafes/<id>	<b>Update</b> - Update a specific cafe
DELETE	/cafes/<id>	<b>Delete</b> - Delete a specific cafe

- **JSON (JavaScript Object Notation):** A lightweight data-interchange format. It's the standard format for sending and receiving data in modern web APIs. Flask has a `jsonify` function to easily convert Python dictionaries to JSON responses.

## Day's Project: Cafe & Wifi API

The goal is to take the cafe data (now stored in a an SQLAlchemy database) and build a RESTful API for it.

### Setup:

- Migrate the cafe data from the CSV file into an SQLite database with an SQLAlchemy `Cafe` model.
- Add a `to_dict()` method to your `Cafe` model to easily convert a `Cafe` object into a dictionary, which can then be converted to JSON.

```
class Cafe(db.Model):  
    # ... columns ...  
  
    #Create a method to convert this object into a dictionary.  
    def to_dict(self):  
        return {column.name: getattr(self, column.name) for column in self.__table__.columns}
```

## Implementing the Endpoints:

```

from flask import Flask, jsonify, render_template, request

# ... App and DB setup ...

# Get all cafes
@app.route("/all")
def get_all_cafes():
    cafes = db.session.query(Cafe).all()
    # Use a list comprehension to convert each cafe object to a dictionary
    return jsonify(cafes=[cafe.to_dict() for cafe in cafes])

# Get a random cafe
@app.route("/random")
def get_random_cafe():
    # ... logic to get a random cafe ...
    return jsonify(cafe=random_cafe.to_dict())

# Add a new cafe
@app.route("/add", methods=["POST"])
def post_new_cafe():
    new_cafe = Cafe(
        name=request.form.get("name"),
        # ... get other data from request.form ...
    )
    db.session.add(new_cafe)
    db.session.commit()
    return jsonify(response={"success": "Successfully added the new cafe."})

# Update a cafe's price
@app.route("/update-price/<int:cafe_id>", methods=["PATCH"])
def patch_new_price(cafe_id):
    new_price = request.args.get("new_price")
    cafe = db.session.query(Cafe).get(cafe_id)
    if cafe:
        cafe.coffee_price = new_price
        db.session.commit()
        return jsonify(response={"success": "Successfully updated the price."})
    else:
        return jsonify(error={"Not Found": "Sorry a cafe with that id was not found."})

# ... Implement DELETE endpoint ...

```

You would test these endpoints using a tool like **Postman** or **Insomnia**, not a web browser (except for GET requests).

## Day 67: Blog Capstone Project Part 3 - RESTful Routing

This day applies the RESTful principles learned on Day 66 to the ongoing Blog Capstone project. The goal is to add functionality to create, edit, and delete blog posts.

### Key Concepts

- **Integrating WTForms with SQLAlchemy:** Creating forms that map directly to the fields of a database model.
- **Pre-populating Forms:** When editing a post, the form fields should be filled with the existing data from the database.
- **Full-stack CRUD:** Connecting a front-end (HTML templates with forms) to a back-end (Flask routes) that performs CRUD operations on a database (SQLAlchemy).

### Project Goal: Add Full Post Management to the Blog

#### 1. Create a New Post ( /new-post ):

- Create a `CreatePostForm` with WTForms for the title, subtitle, author, image URL, and body content. The body should use a `CKEditorField` for rich text editing (requires `Flask-CKEditor` extension).
- The route handles GET (to show the form) and POST (to process it).
- On POST, create a new `BlogPost` object using the form data, add it to the DB session, commit, and redirect to the main page.

#### 2. Edit an Existing Post ( /edit-post/<int:post\_id> ):

- This route uses a dynamic URL to identify the post to edit.
- **GET Request:**
  - Fetch the `BlogPost` object from the database using the `post_id`.
  - Create an instance of the `CreatePostForm`, but pre-populate it with the data from the fetched post object:

```

# In the /edit-post/<post_id> route
post = BlogPost.query.get(post_id)
edit_form = CreatePostForm(
    title=post.title,
    subtitle=post.subtitle,
    img_url=post.img_url,
    author=post.author,
    body=post.body
)

```

- Render the form template, passing in `edit_form`.

- **POST Request:**

- When the populated form is submitted, fetch the post from the DB again.
- Update its attributes with the new data from `edit_form.field.data`.
- Commit the changes and redirect to the post's page.

### 3. Delete a Post ( `/delete/<int:post_id>` ):

- This is a simpler route that only needs to handle GET requests (triggered by a link/button).
- Fetch the post to delete using the `post_id`.
- Use `db.session.delete()` and `db.session.commit()`.
- Redirect to the home page.

```

# In main.py
@app.route("/edit-post/<int:post_id>", methods=["GET", "POST"])
def edit_post(post_id):
    post = BlogPost.query.get(post_id)
    # Pre-populate the form
    edit_form = CreatePostForm(obj=post) # A shortcut for pre-populating

    if edit_form.validate_on_submit():
        post.title = edit_form.title.data
        post.subtitle = edit_form.subtitle.data
        post.img_url = edit_form.img_url.data
        post.author = edit_form.author.data
        post.body = edit_form.body.data
        db.session.commit()
        return redirect(url_for("show_post", post_id=post.id))

    return render_template("make-post.html", form=edit_form, is_edit=True)

```

# Day 68: Authentication with Flask-Login

This day introduces one of the most critical features of any web application: user authentication (registration and login).

## Key Concepts

- **Authentication vs. Authorization:**
  - **Authentication:** Verifying who a user is (e.g., with a username and password). Are you who you say you are?
  - **Authorization:** Determining what an authenticated user is allowed to do (e.g., an admin can delete posts, but a regular user cannot). What are you allowed to do?
- **Password Hashing & Salting: NEVER** store passwords in plain text. A password should be run through a one-way hashing algorithm (like SHA-256) to create a hash. A "salt" (a random string) is added before hashing to prevent rainbow table attacks. The `werkzeug` library (a dependency of Flask) has helpers for this.
- **Flask-Login:** A Flask extension that handles the common tasks of user session management: logging users in, logging them out, and remembering them between sessions.

## Implementation Steps

### 1. Create a User Model:

- This model must inherit from `db.Model` and `UserMixin`. `UserMixin` provides default implementations for the properties that Flask-Login expects user objects to have (e.g., `is_authenticated`).

```
from flask_login import UserMixin

class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(100))
    name = db.Column(db.String(1000))
```

### 2. Set up LoginManager :

```

from flask_login import LoginManager

# ... after app creation ...
login_manager = LoginManager()
login_manager.init_app(app)

# This user_loader callback is used to reload the user object
# from the user ID stored in the session.
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

```

### 3. Registration Route ( /register ):

- Create a registration form.
- When the form is submitted, hash and salt the password from the form.
- Create a new `User` object with the hashed password and save it to the database.

```

from werkzeug.security import generate_password_hash, check_password_hash

# In the register route after form validation
hashed_and_salted_password = generate_password_hash(
    form.password.data,
    method='pbkdf2:sha256',
    salt_length=8
)
new_user = User(
    email=form.email.data,
    password=hashed_and_salted_password,
    name=form.name.data
)
db.session.add(new_user)
db.session.commit()

```

### 4. Login Route ( /login ):

- Create a login form.
- When submitted, first find the user in the database by their email.
- Then, use `check_password_hash()` to compare the stored hashed password with the password the user just typed in.
- If they match, use the `login_user()` function from Flask-Login to create the session.

```

from flask_login import login_user

# In the login route after form validation
user = User.query.filter_by(email=email).first()
if user and check_password_hash(user.password, password):
    login_user(user) # This handles the session.
    return redirect(url_for('secrets'))

```

## 5. Protecting Routes:

- Use the `@login_required` decorator on any route that should only be accessible to logged-in users. Flask-Login will automatically redirect unauthenticated users to the login page.

```

from flask_login import login_required

@app.route('/secrets')
@login_required
def secrets():
    return render_template("secrets.html")

```

## 6. Logout Route ( /logout ):

- This is a simple route that calls the `logout_user()` function.

# Day 69: Blog Capstone Project Part 4 - Adding Users

This day fully integrates the authentication system from Day 68 into the blog project, creating relationships between users and their posts.

## Key Concepts

- **Database Relationships (One-to-Many):** A core concept in relational databases. In our blog, one `User` can have many `BlogPosts`. This is modeled in SQLAlchemy using `db.relationship()` and foreign keys.
- **Foreign Keys:** A column (or a set of columns) in a table that establishes a link between the data in two tables. The `blog_posts` table will have an `author_id` column that links to the `id` of the `users` table.
- **Authorization Logic:** Implementing checks to ensure that only the author of a post can edit or delete it.

# Implementation Steps

## 1. Update Database Models:

- **User Model:** Add a relationship to `BlogPost`. This is the "one" side of the one-to-many relationship. The `back_populates` argument links this relationship to the one in the `BlogPost` model.

```
# In User model
posts = db.relationship("BlogPost", back_populates="author")
```

- **BlogPost Model:** Add a foreign key to the `User` table and the corresponding relationship. This is the "many" side.

```
# In BlogPost model
author_id = db.Column(db.Integer, db.ForeignKey("users.id")) # Note: "users" is table name
author = db.relationship("User", back_populates="posts")
```

*You will need to delete your old database file and recreate it for these changes to take effect.*

## 2. Tie Posts to the Logged-in User:

- When creating a new post, you no longer need an "author" text field in the form. The author is the `current_user` provided by Flask-Login.

```
from flask_login import current_user

# In /new-post route
new_post = BlogPost(
    title=form.title.data,
    # ... other fields ...
    author=current_user # Assign the user object itself
)
```

## 3. Display Post Author Information:

- In your templates (like `index.html` and `post.html`), you can now access the author's information through the post object: `post.author.name`.

## 4. Implement Authorization:

- In the routes for editing and deleting posts, you must check if the logged-in user is the actual author of the post.
- A good way to do this is to create an "admin-only" decorator.

```

from functools import wraps
from flask import abort

def admin_only(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        # If id is not 1 then return abort with 403 error
        if not current_user.is_authenticated or current_user.id != 1:
            return abort(403) # Forbidden
        # Otherwise continue with the route function
        return f(*args, **kwargs)
    return decorated_function

```

- Then, apply this decorator to the routes for creating, editing, and deleting posts. (Note: A more robust solution checks `current_user.id == post.author_id` instead of hardcoding an admin ID of 1).

## 5. Conditional Rendering in Templates:

- Use Jinja2 `if` statements in your templates to show "Edit" and "Delete" buttons only if the `current_user` is the post's author.

```

{% if current_user.id == post.author.id %}
<div class="clearfix">
    <a class="btn btn-primary float-right" href="{{url_for('edit_post', post_id=post.id)}}">
</div>
{% endif %}

```

# Day 70: Deploying Your Web Application

The final step: taking the application you've built on your local machine and making it accessible to the world on the internet.

## Key Concepts

- **Production vs. Development Server:** The Flask development server ( `app.run()` ) is not suitable for production. It's not secure, stable, or scalable. A production-grade Web Server Gateway Interface (WSGI) server like **Gunicorn** is used instead.
- **Cloud Hosting Platforms:** Services like Render, PythonAnywhere, Railway, or Google Cloud that provide the infrastructure to run your web application. (Historically, Heroku was the focus of this

lesson, but its free tier was discontinued).

- **Environment Variables:** A way to manage configuration settings (like secret keys, database URLs, API keys) outside of your code. This is crucial for security. You should never commit secrets to version control (Git).
- **requirements.txt** : A file that lists all the Python packages your project depends on. The hosting platform uses this file to install the necessary libraries. You can generate it with  
`pip freeze > requirements.txt` .
- **Procfile** : A file (specific to platforms like Heroku/Render) that tells the hosting service what command to run to start your application. For a Flask app, this will be a Gunicorn command.

## General Deployment Steps (using a platform like Render)

### 1. Prepare Your Application:

- Make sure your `SQLALCHEMY_DATABASE_URI` is configured to use an environment variable. The platform will provide a URL for its own PostgreSQL database.

```
# Use the provided DATABASE_URL or fall back to a local sqlite DB
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get("DATABASE_URL", "sqlite:///blog.db")
```

- Generate `requirements.txt` : `pip freeze > requirements.txt` . Make sure `gunicorn` is included in this file ( `pip install gunicorn` ).
- Create a `Procfile` (literally a file named `Procfile` with no extension) in your root directory:

```
web: gunicorn main:app
```

(This tells the server: "This is a web process. Run the Gunicorn server on the `app` object found inside the `main.py` file.")

### 2. Use Git:

- Your project must be a Git repository.
- Commit all your code, `requirements.txt` , and `Procfile` .
- Create a `.gitignore` file to exclude files you don't want to upload, like your virtual environment folder ( `venv/` ) and local database files ( `.db` ).

### 3. Sign up for a Hosting Platform (e.g., Render):

- Create a new "Web Service" and connect it to your GitHub account.
- Select the repository for your blog project.

### 4. Configure the Service:

- The platform will likely detect that it's a Python project from your `requirements.txt` file.
- Set the **Start Command** to `gunicorn main:app` .

- Go to the "Environment" section and add your **Environment Variables** (e.g., `SECRET_KEY` , `DATABASE_URL` ). The platform provides the `DATABASE_URL` for you when you create a database add-on.

## 5. Deploy:

- Click the "Create Web Service" or "Deploy" button.
- The platform will pull your code from GitHub, install the dependencies from `requirements.txt` , and run the start command from your `Procfile` .
- You can watch the build logs for any errors. If everything is successful, you will be given a public URL where your application is now live! ✨