Pawfect Match

🔗 Live Site: https://pawfect-match-b7p8.onrender.com/

🐾 Pawfect Match overview 🐾

When I first read the brief and saw that it had a database focus, I wanted to challenge myself to think differently. Rather than taking a typical approach, I decided to step outside my comfort zone and explore a more creative direction. I began ideating ways to bring all my skills together into one cohesive project, which led me to the concept of building an admin-style portal.

I chose to develop Pawfect Match, a playful and engaging platform centred around pet adoption. As an animal lover, I wanted to create something that not only showcased my technical abilities, such as integrating SQL templates, flash submission forms, and pet management features like "Add Pet" and "Edit Pet" pages, but also carried a meaningful purpose.

The idea of pets without homes is something that deeply resonates with me, so I intentionally designed the platform to be cheerful and uplifting. By creating a vibrant and user-friendly experience, I aimed to bring positivity and joy to a serious topic, making Pawfect Match both functional and emotionally engaging.

🐾 Base template 🐾

My base template acts as the backbone of my Flask web application, providing a consistent structure and design across all pages. It includes essential elements like the header, navigation bar, footer, and links to external stylesheets and scripts.

I use Jinja templating to create a {% block content %}{% endblock %} section, which allows each page to inject its unique content while still inheriting the overall layout. By extending this base template with {% extends "base.html" %}, I can keep my code clean and organised, avoid repetition, and ensure that any design changes I make in one place automatically update throughout the entire site.

This setup makes my site easier to maintain and keeps everything looking and functioning consistently.

🐾 Index.html 🐾

My index page is the main entry point of my Flask app, handled by the @app.route('/', methods=['GET', 'POST']) function. It serves both as a welcoming homepage and a place where users can get in touch.

I created a list of steps in the route, which outlines the adoption journey, finding a pet, reaching out, and setting up a meeting, and I passed this into the template so it can be dynamically displayed in the "How We Work" section. On the HTML side, the page extends base.html and sets the title to "Pawfect Match."

Inside the {% block content %}, I include several modular sections using {% include %}, such as stats, featured pets, FAQs, a random dog fact, and a contact form. The form posts back to the same route, and when a user submits it, I collect the form data, flash a thank-you message, and re-render the page with their input preserved

🐾 About.html 🐾

My About page is designed to tell the story behind Pawfect and share the mission, values, and services that drive the platform. In my Flask route for /about, I define a sections list, which contains structured content for each part of the page, like "Our Mission," "What We Do," and "Our Promise."

Each section includes an image, alt text, a title, and one or more paragraphs, along with a reverse flag to alternate the layout for visual variety. I pass this data to the about.html template, which extends base.html for a consistent layout. Inside the {% block content %}, I use a loop to dynamically render each section using the values from the sections list.

Depending on the reverse value, different class names are applied to switch the image and text alignment. I also include reusable components like the hero banner, stats section (conditionally after the second section), a fun dog fact, and a contact form. If a user submits the form, the route captures their input, flashes a thank-you message, and re-renders the page with the form data preserved.

🐾 Pets.html 🐾

My Pets page is where users can explore all the animals available for adoption, and it's designed to be both user-friendly and admin-functional. When someone visits the `/pets` route, the page pulls all pet entries from the database using SQLAlchemy and displays them in a responsive card layout.

I've built in dynamic filtering using GET parameters, so users can narrow down their search by name, age, breed, or species. On the backend, I adjust the query based on what the user submits, so only relevant pets are shown. Each pet card includes an image, name, breed, and age, and for admin purposes, I added Edit and Delete buttons.

Clicking Edit takes me to a form where I can update that pet's details, and the Delete button removes the pet from the database entirely. I've also enabled POST requests on this route so users can submit

a contact form directly from the Pets page, with their input saved and a flash message confirming submission.

🐾 Add.html 🐾

My Add Pet page is designed to let me easily add new animals to the adoption listings through a clean, dedicated form. When I visit the `/add` route, I'm presented with a custom-styled form that collects important details like the pet's name, species, breed, age, and image filename.

Once I submit the form, the data is captured via a POST request, and a new `Pet` object is created and added to the database using SQLAlchemy. After successfully adding a pet, I flash a confirmation message and redirect to the `/pets` page, where the new listing appears immediately.

🐾 Edit.html 🐾

My Edit Pet page gives me full control to update any pet's profile easily and securely. When I access the `/edit/<pet_id>` route, it loads the current data for that specific pet using its unique ID from the database. This allows me to work directly with real-time information and avoid manual lookup or re-entry.

The form is pre-filled with the pet's existing name, age, breed, species, and image. This makes editing quick and efficient, especially if I only need to make a small change. I've also included the option to upload a new image. If I choose to update the photo, the app checks that the file is in a valid format (like PNG, JPG, JPEG, or GIF) before saving it to the uploads directory and updating the image path in the database.

When I submit the form, the app processes the data using a POST request. It then commits the changes to the database with SQLAlchemy and flashes a confirmation message to let me know the update was successful.

🐾 FAQ.html 🐾

I built my FAQ section using a structured two-column layout. On the left side, I included a heading, a short introductory paragraph, and an illustration to set the tone and explain the purpose of the section. This part gives users a friendly overview and invites them to explore the questions. On the right side, I created the list of frequently asked questions using HTML. Each question is wrapped in a button element, and the corresponding answer is placed in a hidden div underneath.

To make the FAQ interactive, I added a JavaScript script at the bottom of the page. I used `document.querySelectorAll` to target all the buttons with the class `faq-title`, then looped through them using `forEach`. For each button, I attached a click event listener. When a user clicks a question, the script finds the parent FAQ item and toggles the `active` class. At the same time, it removes the `active` class from all other FAQ items, ensuring that only one answer is visible at a time. This gives it a clean accordion-style behaviour.

The default behaviour shows the first FAQ open when the page loads by assigning the `active` class to that item in the HTML.

🐾 Contact.html 🐾

I created the Contact page with a form that lets users send their name, email, message, and agree to terms before submitting. On the front end, the form includes inputs for first name, surname, email, and a message textarea. There is also a checkbox where users must agree to the terms and conditions before sending their message. The form uses the POST method and submits the data to the /contact route.

On the backend, I defined a Flask route /contact that handles both GET and POST requests. When the page is loaded with a GET request, it simply renders the contact form template. When the form is submitted (POST request), the route collects the form data from the request. It extracts the first name, surname, email, message, and checks if the user agreed to the terms by verifying if the checkbox was checked.

I then create a new Submission object with the submitted data and add it to the database session. After saving the data, I use Flask's flash function to display a thank you message back to the user. Finally, the page redirects to itself to show the contact form again, now with the success message displayed.