CS373: Software Engineering

10 am, Group 3 - Pets4me Phase III

Connor Sheehan, Rosemary Fortanely, Cristian Garza,
Robert Hrusecky, Andrew Cramer, Dean Torkelson

Section 1: Motivation

The motivation behind Pets4Me was the lack of a single aggregator for pets that are up for adoption in the Austin area. There is never a shortage of animals that need homes and families, but finding the perfect pet requires going to many different shelter websites, visiting many in person, or spending time calling multiple sources for information. However, with Pets4Me, all the data a future owner needs (be it on the shelters, different breeds, or the animals themselves) is presented in one single location, making it easier for animals in need to find their forever home.

Section 2: User Stories

2.1: User Stories for us:

1. **Implement Google-like searching**
    a. As this is a requirement for phase 3, we implemented this. We have a search bar for the whole website, as well as search features for individual model pages.
2. **Make More info Clickable**
    a. This wasn't a requirement, but a good quality of life improvement. On the "Pets" page, we had a link to the pet's info page on Petfinder, but it wasn't hyperlinked. This was an easy change.
3. **Add filtering and sorting to model pages**
    a. As this is a requirement for phase 3, we implemented this. We allow users to specify filters and a sort type and direction in a menu to the left of the grid of instance cards, and these are applied when they click the "Submit" button.
4. **Media on the Shelter instance page**
    a. This was an unfortunate artifact of running out of time last phase - we had to remove the map on the shelter page since it was causing strange test failures. We improved it this time around, so now there's a picture of the shelter logo and the shelter's location on the map.
5. **Instance Page Layouts**
    a. Another good QoL improvement. We had some inconsistent styling on our instance pages. We fixed it so that they all look pretty and relatively similar.

2.2 User Stories for CrashSafe:

1. **Scroll bar on Car Models page doesn't work when page is at reduced width**

    a. To replicate: as a user, if you go to the Car Model model homepage, there's a scroll bar at the bottom of the table that allows you to scroll back and forth. As a user, this works when the page is full-size, but clicking and dragging it doesn't work when the page is at half-width, for example. As a user, I would like for the scroll bar to work regardless of screen width.
2. **Invalid URLs should be handled gracefully.**
    a. As a user, I think it is important that URLs that specify an invalid path to a page on your site should be handled gracefully. To reproduce this issue, visit this example broken link, and notice the navbar is displayed along with a white screen. I think a 404 error page would be nice here, or at least have it redirect the user back to the home page.
3. **Search Highlighting**
    a. As a user, I would like to see the search term highlighted in the results after using your google-like search. This would help to determine why a result is being returned and help to refine my search term. It might be especially useful as your models have many attributes which could cause confusion when searching.
4. **Clicking on car brands from the car model pages**
    a. When on the car models homepage, 'crashsafe.me/carmodel', it would be nice to be able to click on the car brand under the 'MAKE' field, and have it link to that brand's corresponding page. Currently, the only links on the page are to the car models, and from that link you can click the link to the car brand. However, removing the extra click, and having the link on the car model homepage would make for a better user experience.
5. **Fix the NaNs in the filters on the car brands page**
    a. Currently, the filters on the car brands page show a lot of NaNs for the default values. As a user, I want to know the min and max values for the information I'm filtering. This will make sorting and filtering car brands a more intuitive process.

Section 3: RESTful API

GET /filter

This endpoint will return all data needed to populate filter menus.

GET /models/pets

This endpoint will return all instances of individual pets.

GET /models/breeds

This endpoint will return all instances of breeds.

GET /models/shelters

This endpoint will return all instances of shelters.

GET /models/pets/{id}

This endpoint will return the pet that has the id specified by the parameter.

GET /models/breeds/{id}

This endpoint will return the breed that has the id specified by the parameter.

GET /models/shelters/{id}

This endpoint will return the shelter that has the id specified by the parameter.

We chose these endpoints based on a recommendation from a TA on Piazza.

Section 4: Models

For our three models, we decided on 1) the individual pets up for adoption, 2) the different breeds of dogs, 3) breeds of cats, and 4) the shelters where you could adopt the animals.

Model 1: Pets

The Pets model contains the "dog-ographic" information on the individual animal. We included information about the pet itself, such as name, breed, size, gender, and age. Each pet model also contains information about the adoption, such as where the animal currently is and a link to the shelter's website. From a pet's page, you can be linked to the page for the breed, as well as the page on our website for the shelter.

Models 2 & 3: Dog and cat breeds

The Breeds model contains the information that someone who has never heard of it would need to get an idea of what the breed is like, such as its name and nicknames, species, origin, average lifespan, size, typical traits, and how heavily it sheds. From each breed page, you'll be shown a few links to animals that are up for adoption that match that breed, as well as the shelter that has

the most animals of that breed. Though breeds are conceptually very similar, we have them split between dogs and cats in our code due to major differences in the individual attributes for each.

Model 4: Shelters

The Shelter model has essentially every piece of relevant information that someone looking to adopt would need. Including its location, available hours, contact information like emails and phone numbers, adoption policy and mission statements, and of course their social media. If there is any information that someone is looking for that we don't have, there is a link to the shelter's website on the page.

Section 5: Tools

The primary tools we used were React, TypeScript, Marvel, Postman, Bash, Mocha, Selenium, and Mapbox. Within React, we also used react-bootstrap and Material UI, as these both have a similar look and feel and allow us to easily implement complicated components. We chose to work in TypeScript as opposed to JavaScript to reduce the likelihood of type errors and silly human mistakes. We used Marvel to create mockups of the frontend for us to model our designs after. Postman was used to design our API. Bash was used to deploy our website. Selenium and Mocha were both used to test our frontend. Lastly, Mapbox was used to display a map of a shelter's location on its instance page.

Section 6: Hosting

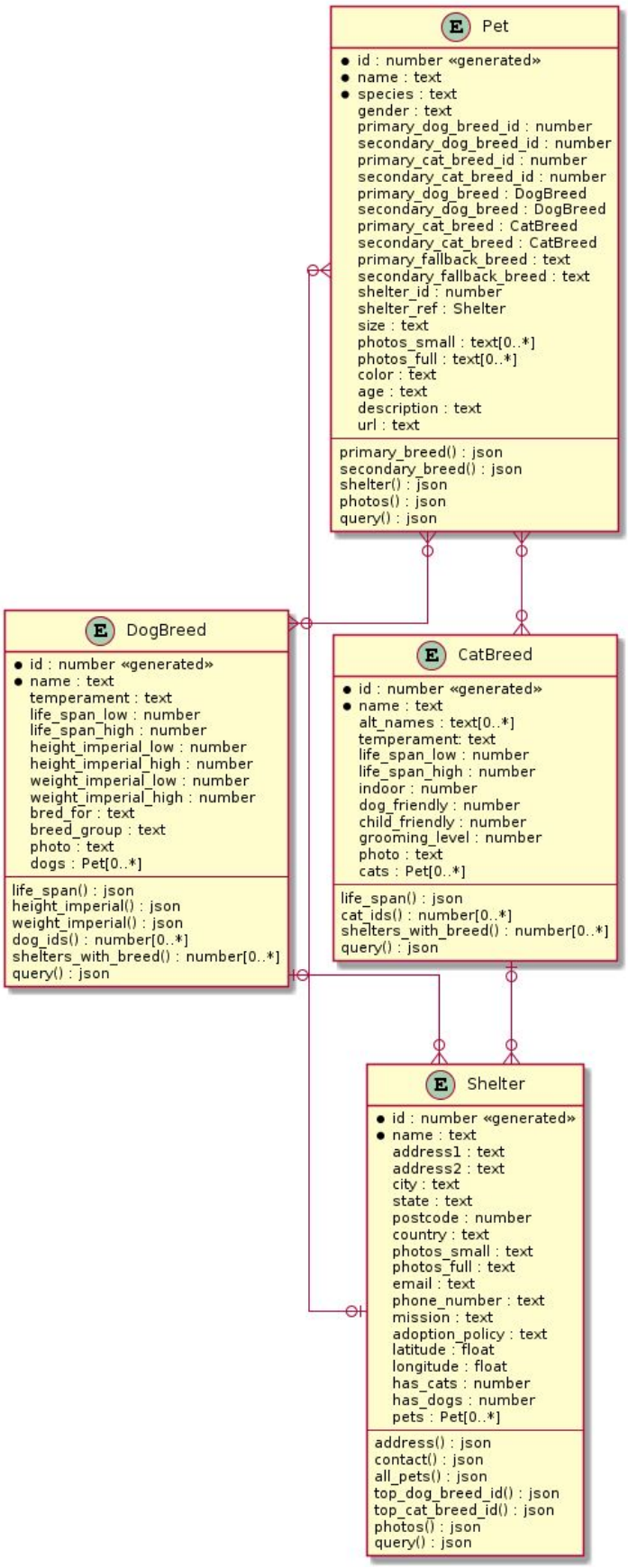
We decided to host our website through GCP. We had the option of hosting through the Compute Engine and the App Engine of which we chose the latter. Although it was more difficult to set up (in our opinion, as with compute engine all that was needed was to start the server process in the VM), App Engine is free as long as your app stays within quotas set by Google. Also, App Engine provides more scalability features and it is easier to re-deploy once the project is set up. We got our domain name through NameCheap. HTTPS certificates are managed automatically through GCP App Engine by a feature known as "Google-managed SSL." This is advantageous because we no longer needed to know how to set up HTTPS or renew certificates manually. We found that the frontend instance needs the API key for our GitLab repository to aggregate statistics on the about page.

Section 7: Pagination

On our website, we implemented pagination as a query parameter to the backend that returns a certain quantity of data after a certain offset. This way, users only see 12 cards at a time, but they can load more if they want to. We not only let users go forward and backward one page at a time, but we show them the total number of pages, and let them go +/- 2 pages (boundaries permitting) at a time, or to the first and last.

Section 8: Database

Our database is hosted in a Google Cloud Platform PostgreSQL instance. We constructed the definitions in python using SQLAlchemy. There are four tables: Pet, DogBreed, CatBreed, and Shelter. Each contains the data, methods, and connections outlined in the UML diagram below. The connections for Pet are a primary and secondary dog breed, as well as a primary and secondary cat breed, and the shelter it belongs to. The connections for Dog Breed are a list of pets that belong to that breed group, and a list of local shelters with that breed. The connections for Cat Breed are a list of pets that belong to that breed group, and a list of local shelters with that breed. The connections for Shelter are a list of pets that it contains, and the dog breed and cat breed that occur the most at that shelter.


Pet
id : number «generated»
name : text
species : text
gender : text
primary_dog_breed_id : number
secondary_dog_breed_id : number
primary_cat_breed_id : number
secondary_cat_breed_id : number
primary_dog_breed : DogBreed
secondary_dog_breed : DogBreed
primary_cat_breed : CatBreed
secondary_cat_breed : CatBreed
primary_fallback_breed : text
secondary_fallback_breed : text
shelter_id : number
shelter_ref : Shelter
size : text
photos_small : text[0..*]
photos_full : text[0..*]
color : text
age : text
description : text
url : text
primary_breed() : json
secondary_breed() : json
shelter() : json
photos() : json
query() : json
DogBreed
id : number «generated»
name : text
temperament : text
life_span_low : number
life_span_high : number
height_imperial_low : number
height_imperial_high : number
weight_imperial_low : number
weight_imperial_high : number
bred_for : text
breed_group : text
photo : text
dogs : Pet[0..*]
life_span() : json
height_imperial() : json
weight_imperial() : json
dog_ids() : number[0..*]
shelters_with_breed() : number[0..*]
query() : json
CatBreed
id : number «generated»
name : text
alt_names : text[0..*]
temperament: text
life_span_low : number
life_span_high : number
indoor : number
dog_friendly : number
child_friendly : number
grooming_level : number
photo : text
cats : Pet[0..*]
life_span() : json
cat_ids() : number[0..*]
shelters_with_breed() : number[0..*]
query() : json
Shelter
id : number «generated»
name : text
address1 : text
address2 : text
city : text
state : text
postcode : number
country : text
photos_small : text
photos_full : text
email : text
phone_number : text
mission : text
adoption_policy : text
latitude : float
longitude : float
has_cats : number
has_dogs : number
pets : Pet[0..*]
address() : json
contact() : json
all_pets() : json
top_dog_breed_id() : json
top_cat_breed_id() : json
photos() : json
query() : json

Section 9: Testing

9.1: Selenium GUI Testing

We used Selenium to test various components of our GUI, such as ensuring that links (model pages, “adopt a pet today” button) take users to the appropriate location.

9.2: Mocha Frontend Testing

We used Mocha to test the logic on our frontend. Our tests range from basic rendering-checks and logic validation to more complicated tests where we verify that certain parts of the page load correctly when we receive bad or incomplete data from the APIs we call.

9.3: Postman API Testing

We used Postman’s testing suite to ensure that our API responds to normal HTTP calls and returns expected data.

9.4: Python unittest Backend Testing

We used Python’s built-in unittest functionality to test a myriad of things rooted in the backend, mostly API-centered. In addition to our Postman tests, we have some Python tests that ensure our internal API is performing as expected, as well as tests that confirm the liveness of our external API data sources (i.e. we’re not being rate-limited, the websites haven’t died, etc.)

Section 10: Filtering/Sorting

Filtering: Filtering can be done across multiple types of quantitative data in our models. However, some of the quantitative data appears to be qualitative because the APIs present it in a qualitative fashion - for example, a pet’s “age” is not presented numerically, but as either “Baby”, “Puppy”, “Adult”, or “Senior”. Since we have a small number of finite values for these fields, it’s still fairly easy for users to filter by those values. Otherwise, we provide dropdown, multi-select menus for string values (breed, species, etc.) and sliders for numerical values (height, lifespan, distance, etc.). We pass this information to the backend to filter using machinery built in to Flask Restless. To accomplish this, the frontend call has to include the name of the table we’re filtering on, the value to filter by, and an operator that details how to use the value for filtering.

Note: Not many zip codes have shelters. We suggest starting at 78702 to see that filter work.

Sorting: Similar to filtering, we allow the user to sort by a variety of fields on each model in either ascending or descending order. We achieve this by passing a value with the filter information that describes which value we want to sort by, as well as the direction we want to sort in.

Section 12: Searching

We allow the users to search in two different ways from the same search bar - they can either search the entire website from one base search (The “Pets4Me” option), or they have the option to search based on individual models. We display the information for them in a way that matches how we display it on model homepages (with cards) and with the query keywords highlighted for them. For the global search, we split the results up based on model type. To search, simply enter your query and **press enter** on your keyboard.