Technical Report for FindADogForMe

Matthew Zhao, Samarth Desai, Bryan Yang, Keven Chen, Daniel Ho

Motivation

As shelters become increasingly crowded at alarming rates, it's become clear that the adoption process must be optimized. Our intent in building Find a Dog for Me is to go beyond simple data aggregation in facilitating the adoption of dogs from shelters. As such, we have begun work on creating a unified system tying together dogs, dog breeds, shelters, and dog-friendly activities in an attempt to simplify finding available dogs and establish motivation for their adoption.

User Stories

Phase 1

We were tasked to complete the following user stories:

1. Finish the breeds page
2. Finish the activities page
3. Finish the shelters page
4. Finish the about page
5. Add more non-white dogs of pictures

In order for the first three issues to be resolved, the backend team needed to scrape data from a variety of APIs listed in our about page. The backend team used the requests library for Python and wrote code to interact with and obtain data from each API. From there, the backend team used sample calls relevant to the Austin area in order to get useful data from each API and provided it to the frontend team. Their python code can be found in the /backend folder of the GitLab repository.

Using the scraped data, the frontend team inserted the information into their respective model and instance pages. In addition, the frontend team created carousels for each instance page. The carousels were filled with pictures from both the scraped data and third-party sources. The fourth user story involved updating the basic static .html that was formatted in Bootstrap. The biographies were hard-coded in, but the issues/commits are dynamically pulled from a JavaScript script interacting with the GitLab API. The last user story was simply a matter of replacing the placeholder images that were in place at the launch of the site with different images.

We tasked our developers with the following user stories:

1. Implement an accessible navigation menu for the mobile version of the website.
2. Add a section for holding related model instances to each model instance.

3. Make disaster model information more readable (spacing, line breaks, highlights).
4. Add a carousel to the landing page to switch between different cover images.
5. Create a table holding total GitLab statistics in the About page.

See https://gitlab.com/oceanwall/findadogforme/issues/14 for additional details.

Phase 2

We were tasked to complete the following user stories:

1. Use images from dog instances in the homepage slide show.
2. Provide multimedia on instance pages.
3. Put instance page links on the model pages.
4. Update the title and favicon of the website.
5. Provide pagination between the instances on the model pages.
6. Add page buttons to facilitate pagination on the model pages.
7. Adjust the appearance of the about page.

For User Story 1, we were unable to complete it because of the varied resolution and dimensions of the images of dog instances that we scraped from our API sources. Rather than take the risk that we might have a low quality, portrait-image in our carousel (and therefore ruin the landing page's appearance), we chose to rely on high-quality images whose dimensions we knew and trusted.

User Story 2 was completed by adding an image carousel and cards of other related instances to each instance page. However, there are some cases where images for a certain instance were not provided by the API so a carousel is not present. Although this is the extent of instance page multimedia for this phase, in the future we will be adding other multimedia such as maps.

User Story 3 was implemented by adding instance cards of that model to the model page. Each card has a button on the bottom that links to the corresponding instance page.

User Story 4 was a quick fix, solved by altering metadata on the index of our React App, and User Story 7 was resolved by moving our list of API data sources to the bottom of the page, and providing tasteful white space to space out items on the page.

User Story 5 was a bit tricky to approach. A lot of the time spent on this user story was for research and planning as opposed to actual coding. In the end, we had the model pages contain the information on what the current page was on, the pagination menu is able to change said current page, and the API calls handle what was on each page. The only exception for the API calls handling what was on each page was "breeds", for the breeds array was small enough such that it was much more efficient to take in the entire array and determine what showed up via arithmetic.

User Story 6 was really quick to finish. React Bootstrap already has a Pagination component, so the frontend team took it and filled it out the information fed from the model page that housed it. This included the current page, and functions to change the page variable.

We tasked our developers with the following user stories:

1. Showing additional attributes for each of the instance cards on the model page.
2. Fixing mobile appearance bugs.
3. Show multiple instances of each model on the model page.
4. Fixing a fixed navigation bar such that it doesn't block out the top of the page.
5. Use a carousel to highlight relevant images while maintaining a neat structure.

Testing

backend/tests.py

Backend testing, which encompassed testing the seeding portion of the project, was handled by Python's built-in unit test framework and can be found within the backend folder inside tests.py. These tests were mainly concerned with ensuring that the seeding scripts used to build the database were in working order and were correctly building the model objects based off of the responses that we were receiving from the APIs that we were scraping.

Tests two through nine inside this file were primarily responsible for testing methods located inside seeding/dogs_seeding.py; specifically, the get_shelters, build_shelter, get_dogs, and build_dog methods. Tests were written to ensure that get_shelters and get_dogs returned the designated amount of objects and were returning valid data. Additionally, the build_shelter and build_dog methods were tested to ensure that they were correctly populating the Shelter and Dog model objects such that all necessary attributes were being filled in with the correct types. The purpose of these tests was to ensure both the consistency of the data that was being added into the database and that all the guaranteed attributes were indeed being filled in with data.

Tests ten through twelve were responsible for covering the seeding scripts found in seeding/breeds_seeding.py. The methods tested were get_breed_images, get_all_breeds, and build_breed. We tested get_breed_images and get_all_breeds to ensure that they were returning a non-empty list. The build_breed method was tested to ensure that its attributes were being correctly filled with data and not left empty. Not all attributes for the breed model are required to be filled though, so the method only tested the attributes that were required to have a value.

Finally, from test thirteen and onward, the methods located within seeding/activity_seeding.py were tested. The methods tested were get_all_parks, build_park, get_all_eventbrites, get_eventbrite_venue build_event, get_all_meetups, and build_meetup. For get_all_parks, get_all_eventbrites, and get_all_meetups the tests simply ran those methods and checked to see if a non-empty list was returned. Testing for get_eventbrite_venue was done by

getting a sample Eventbrite event and passing it to this method. We then checked for a value to be returned that was not equal to None. For build_park, build_event, and build_meetup the tests attempted to build an activity object from the data returned from either get_all_parks or get_all_meetups and built a sample object using the appropriate build method. Each returned object was tested to ensure that each guaranteed attribute was present with a value that was equal to None.

Postman.json

Testing for our API was done through Postman as well as via the front-end wrapper tests, which utilized Mocha and Chai. Postman was used in order to test responses from our server, which was set up using Flask-Restless, and Mocha and Chai were used in order to test the wrapper functions that were being written to help make it easier for the frontend team to interface with the API. The Postman testing file, Postman.json, can be found in the root directory of the project. The wrapper mirrors the functionality defined on Postman, so the tests found inside Postman.json are very similar to the Mocha tests found in frontend/test/mocha_test_babel.js.

We ended up testing every endpoint defined in our API, but due to issues with how Flask-Restless works, some of our endpoints actually required more than one API call to function as intended. To get around this and to ensure that we could continue to use Postman for testing, we ended up providing URLs in the documentation that mirror the last API call the wrapper functions use to get the final result. This ensures that our Postman tests reflect the true behavior of the endpoints that we define. To restate, the functionality of the endpoints is actually handled by the wrapper functions that we wrote, not by the server itself. The Postman tests ensure that the appropriate data is returned as defined by the documentation. For example, for endpoints such as /dog, /breed/dog, /activity/dog, and /shelter/dog, the tests were written to check that they all return a list filled with Dog objects. This check was done by checking for the attributes associated with a Dog object. This process was repeated for each return type and for all endpoints associated with it.

The Postman tests are run via the GitLab CI, using the command “newman run Postman.json”. Our Postman documentation can be found here (https://documenter.getpostman.com/view/6754951/S11KQJxc), and justification for our Postman documentation (with regard to the front end wrapper functions) can be found here (https://piazza.com/class/jqnxdg4iuw55ln?cid=283).

frontend/test/guitests.js

Website functionality was tested using Selenium webdriver as well as the Mocha and Chai testing framework. This file can be located at frontend/test/guitests.js. The basic functionality of the website was tested one page at a time.

The first set of tests looks at the Home Page and ensures that the carousel and navigation bar exist. It also checks that it's possible to navigate to the About Page from there. We test the About Page to ensure that all the required information is present (description, motivation, etc.) and it also checks that the Gitlab statistics are correctly displayed. One final test for the About Page is to check that the "Tools" section is there and that multiple tools are correctly listed.

The "Dog Page" tests check to ensure that dog cards exist on the page, confirm that the dog instance page correctly loads and that the dog instance page contains related instances as well. The "Activities Page" tests check to ensure that activity cards exist on the page, confirm that activity instance pages correctly load, and also check for the existence of related instances present on the instance page. The "Breeds Page" tests check to ensure that breed cards exist on the page, confirm that breed instance pages correctly load, and check for the presence of related instances on the breed instance page. Finally, the "Shelters Page" tests check that the shelter page correctly loads with shelter cards, confirm that shelter instance page loads correctly, and finally checks for the existence of related instances on the instance page.

Although we had our choice of multiple languages to work with Selenium webdriver in, we ultimately chose to use Node.js, primarily because one of our team members had prior experience in working with Selenium in that language.

frontend/test/mocha_test_babel.js

We used the Mocha and Chai testing framework in order to test the wrapper functions we wrote to make interfacing with the API easier for the frontend team. Each test essentially ensured that the return data was appropriate to the call and that it contained properties relevant to the defined return model.

For the tests regarding getShelter, getDogShelter, getBreedsShelters, and getActivityShelters, each test checked to ensure that the data being returned by these methods contained properties that were inherent to shelters. For getDog, getShelterDogs, getBreedDogs, and getActivityDogs, each test checked to ensure that the data being returned by these methods contained properties that were inherent to dogs. For getBreed, getActivityBreeds, getDogBreed, and getShelterBreeds, each test checked to ensure that the data being returned by these methods contained properties that were inherent to breeds. For getActivity, getBreedAcivities, getDogActivities, and getShelterActivities, each test ensured that the data being returned by these methods contained properties that were inherent to an activity. Finally, for the base getBreed, getDog, getActivity, and getShelter methods, we also checked to see if pagination worked as intended.

Gitlab Continuous Integration

Our GitLab CI, as documented in the .gitlab-ci.yml file, uses the two docker containers that we created (https://cloud.docker.com/repository/docker/oceanbarrier/findadogforme-backend/general and https://cloud.docker.com/repository/docker/oceanbarrier/findadogforme-frontend/general) to run frontend, backend, and Postman tests. To summarize, our frontend tests check the integrity of our frontend wrapper functions for the API, our backend tests check the integrity of our seeding scripts and their interactions with our RESTful API data sources, and the Postman tests ensure that our API is returning correct information.

Frontend

Pagination

The model pages themselves keep track of what page they're on, defaulting to page one. Then, they make API calls with the specified page number and passes down information to some of its children components. The model page's relevant children components for pagination are the cards that take up the majority of the page and the pagination menu that appear at the bottom of the page. Most of the heavy lifting of what determines which things should be on which page is done by the backend, except for the breeds model. The breeds model manually decides what set of breeds should appear on each page via arithmetic.

The response from the backend server from the API calls returns an array of objects based on the current page containing the information necessary for the cards. The model page takes this array and creates/displays cards for each element. It passes down each element to the card constructor via props.

The pagination component itself was simply the default React Bootstrap Pagination component with a few props passed down to it from the model page. It receives the current page the model page is on and a method that allows it to change the model's page variable.

Routing

Routing was done using React Router. A router component will loop through all of the routes listed in order and choose the route that matches the current path. If an exact route is listed the route must match exactly with the path. If the route is not exact, then that route can be chosen if only a portion of the path matches. For example, a route that is not exact to /breeds will be chosen if the URL is /breeds/(some breed id) since the first portion of that URL matches the route.

Basic routing was done in App.js which created routes to the home page, about page, and each of the model pages (/home, /about, /dogs, /breeds, /activities, /shelters). Paths are exact for

the home and about pages. The model pages don't use exact paths in order to have nested routes for instance pages. There is also a route to a page not found component if there is an invalid URL.

The match prop that is automatically passed to the model page component is checked to see if an instance page needs to be rendered. The match prop has information about how a Route path matched the URL. If it was a perfect match, then the model page is rendered. If not, then an instance page of that model needs to be rendered. To do this, a new Route path is returned routing to an instance page component with a URL of /(model page URL)/(instance identifier). The instance identifier for breeds is breed name and the instance identifier for dogs, shelters, activities is a unique id.

Model Pages

The model pages are pretty high up on the hierarchy of the website. They are parents to the pagination components, the card components, and the instance page components. They also handle pagination and nested routing as mentioned earlier. Each model component contains a constructor initializes the state, a method to change the page, a method to make the API call, and the render method.

The render method can be broken up into a few chunks. First, it creates an array of cards based upon the information returned from the API calls. If the API calls haven't been returned yet, then it'll stay as null/empty. Then, it checks the routing path. To determine whether it's rendering the usual model page or a specific instance. If it is the usual model page, then it renders the array of cards in a card deck. Otherwise, it renders an instance page.

The cards are tailor-made for each model. They contain a few snippets of information based on the information passed down from the model page that was received from the API calls. The card component itself is based on the React Bootstrap cards, and heavily uses the grid system for formatting. For their images, they were for the most part also included in said information. However, in the case that it was missing/didn't exist, then we made a placeholder image via a custom component. The placeholder image was simply a solid background with white text.
For further details on how the nested routes and pagination work, check out their respective sections.

The model page, instance page, cards, page menu, and placeholder images all have their own .jsx files and .css files.

Instance Pages

The instance pages are children of their respective model pages and can be directly linked to. Each page contains specific information of the instance, a carousel of images, and cards/links to other related instances. For example, the dog instance page contains images of the dogs, cards of

nearby activities, and links to its breed and shelter instance pages. Note that the cards are the same ones that show up on the model pages so that updating code will be less of a hassle.

The steps that run for each instance page goes as follows:

1. Receive an ID from React-Router URL.
2. Use ID to make further API calls
3. Use response from API calls to generate information, links, and cards

The carousel used for the instance pages is different than the carousel used on the front page and has its own .jsx file.

As of the moment, the methods that make the API calls slightly differ from each other. At first, we used multiple variables to check if each call finished loading information, but later we decided that it was much more streamlined to simply have the API calls happen one after another. We plan to smooth them out so that they all look the same.

Hosting on www.findadogfor.me

The frontend React website is hosted on an AWS S3 bucket, using a Cloudfront custom SSL certificate and Route 53 (in addition to NameCheap to secure the domain name) to provide HTTPS certification and give the website its naming alias.

We used the "npm run-scripts build" command to package the React application into static files, and then hosted said static files on AWS S3.

RESTful API

The sources that we relied on to obtain data for our database were the PetFinder API, TheDogAPI, Dog API, Eventbrite API, Meetup API, and National Park Service API. We used these APIs to seed our database with instances of the Activity, Dog, Breed, and Shelter models, and the seeding code (written in python, and taking advantage of the request module) can be found in **backend/seeding**. Our database uses PostgreSQL, and is hosted on an AWS RDS instance.

Our Restful API server (accessible at https://api.findadogfor.me/) was built using Flask, Flask-Restless, and SQLAlchemy.

- **backend/application.py** contains route handling (/, 404, special API cases), the Flask-Restless initialization code that was used to set up the default url/api/model/{querystring} routes used by the front end API wrapper functions and the actual "application.run" code used to start the Flask application.
- **backend/manage.py** set up database migration from SQLAlchemy, and relied on the database models defined in **backend/application/models.py**.
- **backend/application/__init__.py** initialized both the backend server and the database.

Flask-Restless was used to simplify the generation of API endpoints, as it provided us with the requisite API functionality and allowed us to use custom queries to obtain desired results from a model's table. To accompany this, we also created a frontend wrapper library for our API that handled query construction to simplify the usage of our API for our frontend team.

Routes that we handled through traditional Flask route handling included the index page of our API (to guide users to the Postman documentation), 404 pages, and the "/api/breed/shelter" call, which otherwise would have been too expensive of a call to make through the use of the front end wrapper functions.

Our API server is hosted by an AWS Elastic Beanstalk instance, using a Cloudfront custom SSL certificate and Route 53 to provide HTTPS certification and give our API server its naming alias.

Our RESTful API, designed using Postman and accessible here (https://documenter.getpostman.com/view/6754951/S11KQJxc), provides four API calls for each model; one API call to obtain data about instances of that model, and then three API calls for collecting data about related instances of the three other models. All API calls are GET requests, most of which are processed using Flask-Restless.

API Endpoints and Related Wrapper Functions

Endpoints were designed such that, building off the base URL of (https://findadogfor.me/api), you would first add the name of the primary model (/breeds) to receive general information about instances of that model, and then if seeking information about related instances of other models, you would append the name of that secondary model behind the name of the primary model (for instance, /breeds/shelters returns a list of shelter items that host dogs of a breed parameter).

For the general API calls to obtain overall data about instances of some model, no parameters are required, but for the API calls related to collecting data about related instances of the three other models, parameters to identify a specific instance of the main model (generally an id string) are necessary.

/shelter

This call optionally takes in a specific shelter id or page number and returns a list of shelter items matching the criteria. If no parameters are provided, it returns a paginated list of shelter items. The associated wrapper function is getShelters.

/shelter/activity

This call requires the shelter id for which you want to find nearby activities and optionally takes in the range (in degrees) from the shelter or the page of activity information to request. It returns a paginated list of activity items found near a specific shelter. The associated wrapper function is getShelterActivities.

/shelter/breed

This call requires the shelter id for which you want to find breeds and returns a list of breed items found hosted by a specific shelter. The associated wrapper function is getShelterBreeds.

/shelter/dog

This call requires a shelter id for which you want to find dogs, and optionally takes in the page of information of dogs hosted by the shelter. It returns a paginated list of dog items found hosted by a specific shelter. The associated wrapper function is getShelterDogs.

/dog

This call optionally takes in a specific dog id or page number and returns a list of dog items matching the criteria. If no parameters are provided, it returns a paginated list of dog items. The associated wrapper function is getDog.

/dog/activity

This call requires a dog id for which you want to find activities, and optionally takes in range (in degrees, default 0.5) and page of activity information to request. It returns a paginated list of activity objects that would be suitable for the dog within the specified range. The associated wrapper function is getDogActivities.

/dog/breed

This call requires dog id for which you want to find breeds and returns a list of breed objects that the dog is a species of. The associated wrapper function is getDogBreed.

/dog/shelter

This call requires dog id for which you want to find shelters and returns a list containing the shelter item that hosts the dog. The associated wrapper function is getDogShelter.

/breed

This call optionally takes in the specific breed you want information for and returns a list of breed items matching the specified criteria. If no parameters are provided, it returns a paginated list of all breed items. The associated wrapper function is getBreed.

/breed/activities

This call requires the name of the breed you want activities for and optionally takes the page of activity information you want to request. It returns a paginated list of activity items suitable for dogs of a specific breed. The associated wrapper function is getBreedActivities.

/breed/dog

This call requires the name of the breed for which you want dogs and optionally takes the page of information of dogs of the breed. It returns a paginated list of dog items of a specific breed. The associated wrapper function is getBreedDogs.

/breed/shelter

This call requires the name of breed you want shelters for and returns a list of shelter items hosting a specific breed within the specified range. Up to 6 shelters will be returned. The associated wrapper function is getBreedShelters.

/activity

This call optionally takes the activity id for which you want information and page of results and returns a list of activity items matching the specified criteria. The associated wrapper function is getActivity.

/activity/breed

This call requires the activity id for which you want suitable breeds and returns a list of breed items representing dog breeds that would enjoy the activity. The associated wrapper function is getActivityBreeds.

/activity/dog

This call requires the activity id and optionally takes range (in degrees, default 0.5) from the activity. It returns a list of dog items representing dogs in nearby shelters that would be suitable for this activity within the specified range. This list is capped at 12 entries. The associated wrapper function is getActivityDogs.

/activity/shelter

This call requires an activity id and optionally takes a page of shelter information or range (in degrees, default 0.5) from the activity. It returns a list of shelter items near the activity that match the criteria. The associated wrapper function is getActivityShelters.

Additional information and example requests can be found within the Postman documentation (https://documenter.getpostman.com/view/6754951/S11KQJxc).

Models

UML

A UML diagram describing the relationships between the different models can be found in the GitLab repository here https://gitlab.com/oceanwall/findadogforme/blob/master/uml/models.png.

Essentially, a many-many relationship exists between breeds and shelters, breeds and activities, and shelters and activities, and a one-many relationship exists between both breeds and dogs and shelters and dogs.

Dogs

The Dog model has the following attributes: id, shelter_id, name, breed, age, size, sex, description, image_1, image_2, image_3, and image_4.

The model has connections to breeds, shelters, and activities. The model will link to the breed instance corresponding to the current dog's breed, the shelter instance corresponding to the shelter the current dog is located at, and various activity instances located near the shelter that the current dog is located at.

The public RESTful API source supplying the data for this model is the PetFinder API.

Breeds

The Breed model has the following attributes: name, group, min_height, max_height, min_lifespan, max_lifespan, temperament, min_weight, max_weight, image_1, image_2, image_3, image_4, and is_active.

The model has connections to dogs, shelters, and activities. The model will link to dogs that are adoptable of a given breed, shelters that contain the relevant breed, and activities that are suitable for the breed.

The public RESTful API sources supplying the data for this model are TheDogAPI and Dog API.

Shelters

The Shelter model has the following attributes: id, name, latitude, longitude, city, state, zip code, phone, and address.

The model has connections to dogs, breeds, and activities. The model will link to dogs that the shelter contains, the breeds of those dogs, and dog-friendly activities near the shelter.

The public RESTful API source supplying the data for this model is the PetFinder API.

Activities

The Activities model has the following attributes: id, type, URL, name, description, latitude, longitude, location, is_active, is_free, image_1, image_2, image_3, and image_4. National parks have additional attributes designation, weather, and directions, and meetups have an additional attribute date.

The model has connections to dogs and shelters. The model is linked to dogs that are suitable for this activity, and to shelters that are near each activity.

The public RESTful API sources supplying the data for this model are the National Park Service API, the EventBrite API, and the Meetup API.

Tools

GitLab

GitLab was used as a version control tool, as well as a tracker for various issues. We were able to track the progress of our project based on issue completion and assign various sub-parts of the project to different team members. We also used GitLab CI to run our frontend mocha tests, backend unittest tests, and Postman tests to ensure the continued integrity of our code.

Postman

We used Postman to define and test the endpoints that we've created. While the functionality of these endpoints is actually performed via the wrapper functions we created, we still used Postman to check the validity of the returns of each call. We mimicked the behavior of the wrapper functions by passing in the URL that gets the final result as the URL that Postman uses to test the endpoint. This, coupled with our wrapper function tests, ensures that the API functions as intended.

Bootstrap

Bootstrap was used primarily for its CSS capabilities in styling the HTML files. We relied on its grid layout system and margin/padding CSS shortcuts to format the majority of our static HTML files. We also utilized its JavaScript components to power the navigation bar and the multiple carousels dispersed throughout the website, including on the landing page and on each of the instance pages.

Slack

Slack was the main communication channel used by the team. We utilized a WebHooks integration to receive GitLab event notifications about our repository.

jQuery

jQuery was used by BootStrap to power the carousel and navigation bar.

Docker

Docker images were created for the frontend and backend stacks to run continuous integration scripts, and the backend docker image was also used for backend development in conjunction with the python virtualenv.

Mocha

Mocha was used in conjunction with Chai to run the API wrapper tests and the Selenium GUI tests.

Chai

Chai is an assertion library that was used in conjunction with Mocha to test API wrapper functions and the Selenium GUI tests.

React

React was used as our frontend framework to build user interfaces and implement routing

React-Router

React-Router was used to implement routing for our React App, including routing between different pages (landing page, about page, model pages) and individual instances (model/instanceID).

Selenium

Selenium provided browser automation and was instrumental in allowing us to test our frontend GUI.

Flask

Flask was used to run our API server, and we also utilized its derivative packages, including Flask-CORS (which was used to provide cross-origin resource sharing and make the API accessible to the React frontend) and Flask-Restless (which generated the basic RESTful API used with queries to form the majority of our API endpoints).

Babel

Javascript compiler used to bridge the gap between CommonJS and ES6 import/export syntax with regard to the front end wrapper functions and the frontend mocha tests.

SQLAlchemy

SQLAlchemy provided a generalized interface for creating and interacting with our database using Python instead of SQL statements. It allowed us to define our database models, seed our database, and make database queries in Python.

PostgreSQL

Used to create and manage our relational database.

Hosting

Namecheap

Namecheap was used to obtain the domain name and was instrumental in verifying domain ownership to receive an Amazon-issued certificate for HTTPS.

AWS

AWS was used to host both the frontend and backend of the website. We used an S3 bucket to host our static React files, and an Elastic Beanstalk instance to host our API server. We also utilized Cloudfront custom SSL certificates and Route 53 to provide HTTPS certification and naming aliases to both websites (https://findadogfor.me/, https://api.findadogfor.me/).