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 a 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/12 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.

See https://gitlab.com/oceanwall/findadogforme/issues/25 for additional details.

Phase 3

We were tasked to complete the following user stories:

1. Add website-wide searching accessible through a navbar search bar.
2. Offer alphabetical sorting for the Shelter model instances.
3. Offer filtering of Dog model instances by their specific breed.
4. Offer filtering of Shelter model instances by their city.
5. Offer chronological sorting for the Activity model instances.

User Story 1 was addressed by the backend team by the implementation of four routes on our Flask server to respond to all filtering, sorting, and searching queries regarding each of the four models. These routes are represented by API/shelter/query, API/activity/query, API/breed/query, API/dog/query, and can all be found in backend/routes.py. All of these routes take in a "search" parameter that is checked for in the model's instances' attributes, and the routes for the Activity and Dog models also take in an "include_description" parameter to search within attributes not displayed on the instance cards. Wrapper functions for each of these routes (getShelterQuery, getDogQuery, getBreedQuery, and getActivityQuery) for the frontend team to use were also created to simplify the process of interacting with the API.

The frontend team addressed **User Story 1** by adding an additional '/search' route to a website-wide search page, which displays relevant model-separated results to a navbar search query, and implementing the navbar search bar such that it responds to search queries by redirecting the user to the website-wide search page (populated with results that match the search query).

Because User Stories 2 - 5 all dealt with sorting and filtering, they were addressed by the frontend team through the creation of a query interface specific to each of the model pages. This query interface allows users to select their desired filtering, sorting, and searching parameters and then uses the matching frontend wrapper function to obtain matching results from the API, which are then displayed. Pagination is handled by the backend API wrappers.

User Story 2 was addressed by the backend team by implementing a new route on our Flask server to respond to all filtering, sorting, and searching queries regarding the Shelter model. This route is represented by /API/shelter/query. In addition to adding alphabetical sorting options, reverse alphabetical, zip code and reversed zip code sorting was also added. The shelter_query method in routes.py contains the code used to process a query request and filter the table appropriately to provide the matching instances. A wrapper function by the name of getShelterQuery was also added to facilitate the frontend team in implementing this functionality.

User Story 3 was addressed by the backend team by implementing a new route on our Flask server to respond to all filtering, sorting, and searching queries regarding the Dog model. This route is represented by /API/dog/query and the implementation can be found within the dog_query method inside routes.py. We added the ability to pass in a breed name when making the query API call, and this allowed us to filter the dog instances in our database based on the breed name passed in. A wrapper function by the name of getDogQuery was also written to support the frontend team in implementing this user story on the website.

Similarly to User Story 2, **User Story 4** was addressed by the backend team by implementing a new route on our Flask server to respond to all filtering, sorting, and searching queries regarding the Shelter model. To filter by city, we authenticated all cities that our Activity data source could locate events in, and then passed that data to the frontend team to provide them with a list of valid cities. When a city parameter was passed in with a call to API/shelter/query, we filtered the Activity model table for all instances with a matching city parameter, and then returned those instances. Like User Story 2, the functionality was represented by API/shelter/query route, the shelter_query method in routes.py, and the getShelterQuery frontend wrapper function.

User Story 5 was addressed by the backend team by implementing a new route on our Flask server to respond to all filtering, sorting, and searching queries regarding the Activity model. This route is represented by /API/activity/query and offers alphabetical, reverse alphabetical, chronological, and reverse-chronological sorting. The activity_query method in routes.py contains the code used to process a query request and filter the table appropriately to provide the matching instances. A wrapper function by the name of getActivityQuery was also added to support the frontend team in implementing this user story on the website.

We tasked our developers with the following user stories:

1. Ability to filter states by their population and area.
2. Fix a visual bug associated with small card widths on small screen sizes.
3. Fix a mobile appearance bug associated with the model page title and current page indicator.
4. Provide clickable icons for each of the tools displayed on the About page that link to the tool's website.

5. Clean up the Organization model instance pages, and make social media links more accessible.

See https://gitlab.com/oceanwall/findadogforme/issues/100 for additional details.

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 to nineteen, 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.

For Phase III additional tests were added to ensure that methods such as get_shelters and get_dogs returned the correct number of elements based on the count variable passed in. Different shelter IDs were also passed in to ensure that the methods returned the correct number of results for different shelters. Tests twenty-four through thirty all test the get_breed_images call with different breeds to ensure that the method works across the different acceptable breeds.

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.

For Phase III the Postman documentation was updated to reflect the new searching functionality that was added to our API. We tested /search in a very similar way to the previous endpoints described. Since search can return a mix of any model type, our tests run by checking for attributes that exist for each model type. Our example query performs a search using the string “border collie”, based on our implementation, it will look at every model’s visible attributes and look for matches. In this case, the only matches that exist for this query are the breed model for a Border Collie as well as all the adoptable dogs that are of the same breed. As a result, our tests check for the presence of a breed model as well as a dog model by looking for attributes associated with each type.

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 web driver 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, created in Phase II, 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.

For Phase III additional Selenium tests were added to ensure that the new functionality added during this phase worked as intended. The first new test added was to check that the vector clickables on the home page load correctly and all load in. The next set of tests added were used to check that the embedded Google Maps tool loaded on a sample example instance page from each model. A test was also added to check that the navigation bar remained present across different pages with all the required model page links still there. Many comprehensive tests were also added in to ensure that the searching, sorting, and filtering options available for each model correctly loaded and functioned as intended on the website. Example search queries were passed in using Selenium and each test checked for the presence of specific attributes on the page.

Although we had our choice of multiple languages to work with Selenium web driver 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.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.

For Phase III additional tests were added for each of the previous functions described that further ensured that each of them worked as intended. These tests differ from the ones that were already there by checking different input parameters. To reflect the new functionality added to the API we checked the associated wrapper functions that were written during this phase of the project. These include getDogQuery, getShelterQuery, getActivityQuery, getBreedQuery and getSearchQuery. For each of these tests, we checked if the correct model was returned for each query type (check if getDogQuery returns dog models, check if getShelterQuery returns shelter models, etc.). For each of the query tests, we checked that each of the returned instances was directly related to the parameters passed in. Additionally, for the getShelterQuery and getActivityQuery methods, we also checked if the correct page was returned based on the input. For the getWebsiteQuery tests, we checked if the correct mix of models were returned and that the instances returned were directly related to the input parameters. The second getWebsiteQuery test ensures that the dogs array returned by the API only contains instances directly related to the input parameters. In our case, we check if our input “border collie” only returns dog instances that have the breed attribute of “border collie” or contain a description that refers to “border collie”.

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 the correct information.

Frontend

Our main frontend file is located at frontend/src/index.js (as opposed to frontend/index.js).

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 pass 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 allow for the usage of nested routes with instance pages. There is also a route to a "page not found" (404) 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). For example, 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, cards, and the Google map

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

The Google map component is a npm package that wraps a small corner of the Google Maps API. It uses a key that is currently connected to one of the front end member's GCP server. The map itself is rendered using the latitude and longitude of the instance. For dog and breed instances, it uses the shelter it's housed in and nearby shelters respectively. It places markers that have on-click events. The on-click events change the state of the component, which in turn updates the information window to host the name of the relevant item. Note that there is only one information window in the component, as opposed to one for each marker; the information window moves to display updated information based on the selected marker. Layout wise, the Google Map was rendered in a fixed column for the instances that it takes up the whole width of the screen, while on the other instances it was rendered to fit the dynamic size of the column.

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 during Phase 4's refactoring process 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, uploading them using the S3 command line interface.

Database

The database consists of four separate tables: the Activity, Breed, Dog, and Shelter tables, each of which represents a specific model used in our website. The relationship between the Dog model and the Breed and Shelter models is encoded in the database, as the Dog hosts a shelter_id

foreign key and a breed_name foreign key. This ensures that each dog is both of a valid breed and hosted in a valid shelter. Otherwise, the relationships between all other models are based in geographic proximity (using latitude and longitude) and suitable level of activity. These relationships are not encoded into the database and are rather simulated in the backend server by filtering using the desired parameter.

The database is used by the backend flask server (hosted on Elastic Beanstalk) to provide a RESTful API for the frontend. The backend flask server connects to the AWS RDS database instance and uses Flask-SQLAlchemy to facilitate database operations. The Flask-Restless routes for each model rely on passed query strings matching model attributes to desired values, and the database processes these queries by filtering the respective model table for instances that possess the desired attributes. The filtering/sorting/searching routes also rely on passed-in query parameters. Additional details for this implementation can be found in the Filtering, Sorting, and Searching sections found later in this technical report.

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**. The seeding must take place in a specific order, with Breeds being seeded first, then Shelters, then Dogs (as Dogs have foreign keys rooted in Breeds and Shelters), and finally activities. Our database uses PostgreSQL and is hosted on an AWS RDS instance.

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 one-many relationship exists between (breeds and dogs) and (shelters and dogs), and many-many relationships exist between (activities and breeds), (activities and shelters), (activities and dogs), and (breeds and shelters).

Dogs

The Dog model has the following attributes: id, shelter_id, shelter_name, 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_active_string, is_free, is_free_string, image_1, image_2, image_3, and image_4. National parks have additional attributes designation, weather, and directions, and meetups have an additional attribute for a date.

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

Backend (RESTful API)

Our main backend file is located at backend/application.py (as opposed to backend/main.py).

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

- **backend/application.py** contains the actual "application.run" code used to start the Flask Application.
- **backend/routes.py** contains all route handling for the backend API. This includes the query API routes (used to facilitate filtering, searching, and sorting for each model and for the entire website), the Flask-Restless default URL/API/model/{querystring} routes (both of which are used by the front end API wrapper functions), special case API functions (that require user implementation rather than Flask-Restless implementation to improve response time), and the 404 error handler.
- **backend/utilities.py** contains functions used by route handler methods in routes.py in order to avoid redundant code.
- **backend/manage.py** sets up database migration from SQLAlchemy, and relies on the database models defined in **backend/application/models.py**.
- **backend/application/__init__.py** initializes 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.

We also implemented the query routes via traditional Flask route handling. These functions used to fulfill the searching, sorting, and filtering requirements had to be manually implemented in order to implement the database interactions necessary to query out matching instances. Pagination was manually added in to control the flow of results and mimics the same pagination used in the Flask-Restless auto-generated API methods (so 12 activities and shelters at a time, 20 dogs at a time, and all matching breeds at once). Additional details can be found in the Filtering, Sorting, and Searching sections found below.

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 general 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. These API calls are GET requests and are mostly processed using Flask-Restless. We also provide one API call per model for querying each model (searching, sorting, and filtering based on passed parameters), and a general API call that searches through each instance in each model, including attributes that are not displayed on the cards.

Filtering

Backend Implementation

Each model has a query route hosted on our Flask-based API server that takes in three query string parameters demarcating the three attributes to filter by. If no parameter for an attribute filter was provided (thus providing the parameter with a default value of None), then filtering for that attribute was ignored. Flask-SQLAlchemy was used to filter the models and perform database actions.

Activities can be filtered by whether or not they're high-activity (boolean), whether or not they're free (boolean), and the type of activity (string). Filtering for these parameters was a simple matter of comparing whether the attributes of some given instance matched the desired attribute value (using Flask-SQLAlchemy's "filter_by" function), and keeping it if it satisfied the attribute requirements.

Breeds can be filtered by their group (string), their lifespan (integer), and their height (integer). With regard to lifespan and height filtering, a breed is returned if the provided lifespan and/or height is within the range for that breed. Filtering for groups relied on directly matching attributes (using Flask-SQLAlchemy's "filter_by" function), whereas filtering for lifespan and height checked using Flask-SQlAlchemy's "filter" function that the desired attribute was within the range specified by the specific instance (for example, an instance has a min-height of 15 and a max height of 20, so all values 15 to 20 would include that instance when filtering).

Dogs are filterable by age (string), size (string), and breed (string). Valid values for these attributes are dictated by the database and were communicated to the frontend team to limit the available choices. Filtering for these parameters was similar to filtering for Activities, as it only involved matching passed-in parameters to the instances' attributes (using Flask-SQLAlchemy's "filter_by" function).

Finally, shelters are filterable by zip code (integer), phone area code (number), and city (string). Zip code and city filtering was accomplished by simply matching the passed-in parameters to the instances' attributes (using Flask-SQLAlchemy's "filter_by" function), and phone area code

filtering was performed by checking if the instances' "phone" attribute started with the provided area code (using Flask-SQLAlchemy's "startswith" function).

Frontend Implementation

Whenever a filter is specified, the frontend makes a wrapper function call using the filters to the backend API and the list containing the cards to be displayed in the state of the model component is changed to contain the cards that match the filter.

The shelter model page is filterable by zip code, phone area code, and city. The inputs for zip code and phone area code are taken in by two separate forms. The length of the input for the zip code is capped at five and the length of the input for area code is capped at three. The filter initiates when the input of the form reaches the cap limit or is empty. The city filter is implemented by a typeahead search with autocomplete. The user can type to search for a city and the user will be presented options they can select via a drop-down window as they type. The city filter is initiated when the user presses enter while typing in the filter box or clicks on an option in the drop-down window. The options for the city filter are all the cities contained in the API.

The dog model page is filterable by age, size, and breed. The options for age are baby, young, adult, and senior. The options for size are small, medium, large, and extra large. The options for the breed filter are all the breeds contained in the API. The inputs for age and size are taken in by dropdown buttons. The user can click on the dropdown button and the filter is initiated when the user clicks on an option. The breed filter is implemented by a typeahead search with autocomplete.

The breed model page is filterable by group, lifespan, and height. The options for the group filter are all the dog groups contained in the API. The range of acceptable numbers for lifespan is 6-20, and the range of acceptable numbers for height is 8-35. Given a height or a lifespan, the page will show all dogs whose height ranges or lifespan ranges contain the filter value. All three of these filters are implemented using a typeahead search with autocomplete.

The activity model page is filterable by intensity, cost, and type. Intensity can either be active or casual. The cost can either be free or paid. Type can be Eventbrite, Meetup, or National Park. Each of these filters is implemented by a dropdown button and the filter is initiated when a user clicks on an option.

Sorting

Backend Implementation

Each model has a query route hosted on our Flask-based API server that takes in one query string parameter demarcating the attribute to sort by. The valid sorts for each model are defined beforehand and were communicated to the frontend team in order to restrict what sort preferences are passed to this query route. Flask-SQLAlchemy was primarily used to sort the instances, although a custom sort function was implemented to sort dogs by size.

All models could be sorted alphabetically (ascending or descending order), and this was facilitated by using the “order_by” function provided by Flask-SQLAlchemy on each instance’s name, and demarcated by passing in either “alphabetical” or “reverse_alphabetical” as the string value to the routes’ “sort” param. For all models, in order to sort in descending order, the .desc() function was added to the end of the passed in attribute used by “order_by” to sort the instances (IE: order_by(Model.attribute.desc())).

The Activity model offers sorting by date (chronological and reverse chronological order, represented by the “date” and “reverse_date” strings respectively), and this was done by first filtering out all activity instances without a date attribute (mostly national parks) using Flask-SQLAlchemy’s “filter” function, and then directly ordering by using the date attribute in conjunction with Flask-SQLAlchemy’s “order-by” function.

The Breed model offers sorting by group (alphabetical and reverse alphabetical order, represented by the “group” and “reverse_group” strings respectively), and this was done by simply using Flask-SQLAlchemy’s “order_by” function in conjunction with “group” attribute of the Breed model.

The Dog model offers sorting by size (increasing and decreasing order, represented by the “size” and “reverse_size” strings respectively), and this required the creation of a custom function to order the sizes. Because the sizes are not ordered alphabetically (S, M, L, XL), a function was created mapping larger sizes to larger values and used as the key in order to sort a collection of valid dog instances that had already undergone filtering by breed, age, and size. To sort the sizes in reverse order, the “reverse” parameter to the sort method was set to true.

Finally, the Shelter model offers sorting by zip code (increasing and decreasing order, represented by the “zipcode” and “reverse_zipcode” strings respectively), and this was also done by simply using Flask-SQLAlchemy’s “order_by” function in conjunction with “zipcode” attribute of the Shelter model.

Frontend Implementation

Each model page can be sorted alphabetically or reverse alphabetically. In addition, each model page has a unique additional way to sort them.

Shelters can be sorted by zip code, dogs by size, breeds by group, and activities by chronological order. In order to select the sort order, the user can click on a drop-down box labeled as sort and select one of the options. Once clicking on the option, the model page proceeds to make a query call to the backend server and updates what is shown accordingly. The sorting itself is not handled in the frontend side of the app.

Searching

Backend Implementation

Each model has a query route hosted on our Flask-based API server that takes in one query string parameter demarcating some string to search for. Searching takes place after all instances have been filtered and sorted according to relevant parameters, and only checks against visible attributes (attributes on each of the instance cards) unless a specified parameter (include_description) is set to true, upon which non-visible parameters (such as descriptions) are also checked. No special tool was used to check for a search term being within some attribute; rather, an if-statement block checking all of some instance's attributes utilizing Python's "in" operator was used. As stated prior, results are paginated according to the general pagination scheme set by the Flask-Restless default API routes.

Searches for activities are performed by taking each instance, serializing it into a python dict, and then searching for the search parameter within the activity's name, date, designation, cost, level of activity, location, and type. If the "include_description" parameter is included, then the activity's description is also checked for the search parameter. If the search parameter is found in any of these attributes, then the instance is returned.

Searching for breeds is performed by taking each instance, serializing it into a python dict, and then searching for the search parameter within the breed's name, group, temperament, min and max lifespan, and min and max height. If the search parameter is found in any of these attributes, then the instance is returned.

Searching for dogs is performed by taking each instance, serializing it into a python dict, and then searching for the search parameter within the dog's name, shelter, breed, age, and size. If the "include_description" parameter is included, then the dog's description is also checked for the search parameter. If the search parameter is found in any of these attributes, then the instance is returned.

Searching for shelters is performed by taking each instance, serializing it into a python dict, and then searching for the search parameter within the shelter's name, city, zip code, phone, and address. If the search parameter is found in any of these attributes, then the instance is returned.

Our original plan to implement sitewide searching was to create a specific API query route (API/search_website) that, given a specific search parameter, would search all attributes of all instances of all models, and return all matching instances. However, after extensive discussions between the frontend and backend teams, we ultimately decided to utilize the existing models-specific query functions to fulfill site-wide searching. Not only could they easily be adapted to search for all of a model's attributes, but they also already had pagination set up appropriately.

As such, sitewide searching only utilizes the four previously mentioned query methods. Usage of these query methods is done by only passing in a search parameter, and for the Activity

and Dog models, the “include_description” parameter. This is because these models have additional information associated with them not displayed in their instance cards, and as defined on Piazza, the site-wide search needs to be able to search through information associated with instance pages as well.

Frontend Implementation

The model page searches utilize an input box in order to capture what the user wants. Once the user presses enter or the search button, the text is used as a search parameter in the query to the backend server. After receiving a response, the model page renders the results as cards. In addition, it passes to each of the cards the search parameter, which the cards use to highlight any text that matched to it. The highlighting is done through a package from npm.

The global search functionality was implemented via a new component and adding a search bar in the navbar. The navbar uses the withRouter function in order to inherit the history from the App. When the user pushes the search button or presses enter, the navbar changes the URL, which in turn renders the new component. The new component uses the parameters in the URL in order to create the search query for each model. It then displays the results as cards, organized by tabs.

The site-wide search differs slightly from model page search for the 'Dogs' and 'Activities' models, as the site-wide search scans descriptions for matching characters whether or not they are displayed on card (in order to capture all relevant results). Search implemented for the model page only checks text displayed on the cards themselves and highlights every character match found.

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.

The query API calls used for filtering, searching, and sorting tasks follow the same design (API/ModelName/query?param1=__&param2=__). Each model has only one query API call associated with it, but this query API call controls filtering, searching, and sorting for all instances of that model. The parameters available for these query API calls include parameters to filter by, sort by, and search by, and page numbers. For the Dog and Activity models, an additional parameter

“include_description” allows for searches to include non-card attributes, and this parameter is only used when calling the function as part of a website-wide search.

These API endpoints can be found in backend/routes.py.

/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.

/shelter/query

This call filters a list of shelters based on various filters and sorts passed in. Permissible filters include sorting by city, zip code, and phone number associated with the shelter. It is also possible to order the returned shelters by alphabetical, reverse alphabetical, zip code or reversed zip code order. All results are paginated and the desired page number may also be passed in. All filter, sort, and page parameters are optional. Inputting no parameters will return an unordered list of all shelters, paginated. The associated wrapper function is getShelterQuery.

/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.

/dog/query

This call filters a list of dogs based on various filters and sorts passed in. Permissible filters include sorting by breed, age, and size of the dog. It is also possible to order the returned dogs by alphabetical, reverse alphabetical, size or reversed size order. This call also includes an optional flag that allows you to look for matches within the description of each dog. This flag is false by default. All results are paginated and the desired page number may also be passed in. All filter, sort, and page parameters are optional. Inputting no parameters will return an unordered list of all dogs, paginated. The associated wrapper function is getDogQuery.

/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 the 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.

/breed/query

This call filters a list of breeds based on various filters and sorts passed in. Permissible filters include sorting by group, lifespan, or height associated with the breed. It is also possible to order the returned breeds by alphabetical, reverse alphabetical, group or reversed group order. Results for this query are not paginated and an option to get a specific page is not available. All matching results are returned at once. All filter and sort parameters are optional. Inputting no parameters will return an unordered list of all breeds. The associated wrapper function is getBreedQuery.

/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.

/activity/query

This call filters a list of activities based on various filters and sorts passed in. Permissible filters include sorting by the level of activity, price, or type. It is also possible to order the returned breeds by alphabetical, reverse alphabetical, date or reversed date order. This call also includes an optional flag that allows you to look for matches within the description of each activity. This flag is false by default. All results are paginated and the desired page number may also be passed in. All matching results are returned at once. All filter, sort, and page parameters are optional. Inputting no parameters will return an unordered list of all activities, paginated. The associated wrapper function is getActivityQuery.

/search

This call takes in a string and searches across every instance in the database and looks for matches within the visible attributes for each instance. The string you use to search with is a required parameter. The call will return any mix of instances across any of the models. Additionally, all results are paginated and the only optional parameter for this call is the page number you would like.

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

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. The ability to create and reuse JSX components was instrumental in providing users with a consistent and dynamic user interface.

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), (search/searchParam). It was also used in conjunction with the react-router-bootstrap package to implement highlighting the current page in the navigation bar.

Selenium

Selenium provided browser automation and was instrumental in allowing us to test our frontend GUI. We used the “selenium-web driver” NPM package in addition to ChromeDriver and GeckoDriver to implement the automated GUI tests.

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 to allow the React frontend to access the API) and Flask-Restless (which generated the basic RESTful API used with queries to form the majority of our API endpoints). Flask-SQLAlchemy was also instrumental when

coding database interactions, such as in seeding the database or filtering instances of certain models based on provided filter, search, and sort parameters.

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. It was especially important in implementing the required filtering, sorting, and searching functionality for Phase III, as it acted as a simplified interface to interact with our group's database. For more information, please refer to the above sections "Filtering", "Searching", and "Sorting".

PostgreSQL

Used to create and manage our relational database. Four tables (one for each Model) were created), with a database-defined relation between the Dog, Breed, and Shelter models.

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/).