Doctor Search Technical Report

Abhinav Kasamsetty, Nathan Hoang, Derrin Ngo, Sitong Li, Rohan Arya

Motivation

Currently, there aren't many good ways to search for doctors on the internet. People can search for doctors in their area on Google, but the results are not all-inclusive and only show the closest results. Because of this issue, we came up with the idea for Doctor Search, a website where users can search for all doctors in major cities and learn what specialties are being practiced in them to make an informed decision on which doctor to go to.

Phase 1

Tasks

1. Create a website about doctors, cities, and specialties and an about page.
2. Create Postman API documentation.
3. Get a domain and add SSL support to the website.

To resolve the first task, we used React and NodeJS and constructed a website. We manually found data from our data sources and entered information about doctors, cities, and specialties into JavaScript arrays, which were then used as data in the React application to form the components on the pages. To improve the look of the website, we used the Material-UI CSS framework. On the about page, we manually entered information about biographies and tests but used the GitLab API to dynamically generate each member's issue and commit count.

For the second task, we used the Postman desktop application to construct our schema. We wrote an API schema using the OpenAPI 3.0 standards and generated a collection and documentation from the schema. The schema can be found in `backend/api-schema.yaml`. More information about the API is in the RESTful API section below.

To complete the last task, we obtained the domain doctorsearch.me from Namecheap for free, where our website is currently at. We added SSL support by changing our DNS provider from the domain to AWS Route 53 and using AWS Amplify to add a domain to our website, which automatically handled SSL certification.

User Stories

Customer User Stories

We were tasked with the below:

1. **Rating Scale** - What are the ratings for the doctors out of? 5? 10? I think it would be more clear if you had '4.6/5' instead of '4.6'.
2. **Title** - When you look at the title (in the tab with the favicon), SEARCH is capitalized. Does search stand for something? If not, maybe change it to uppercase for consistency.
3. **Cities Zip Code** - I'm assuming the first string of digits represents the zip code for each city card. I think it would be more clear if you explicitly state that as a zip-code as well as to keep the consistency.
4. **Whitespace** - There seems to be a lot of white space when you click on additional info, maybe you can have the information and location side by side to help fill this in?
5. **Center titles on model pages** - I think it would be cleaner if you could center the titles on the model pages since your card components are already aligned in the center.

The first three user stories were easy to resolve as they were quick changes to the text in the React application. However, the fourth user story was more difficult as it required us to use the `Grid` component from Material-UI on the instance pages with one column for the information and one for the map. The last user story required changing some styling of the text in our Material-UI `Card` component for each model.

Developer User Stories

We tasked our developers, CostlyCommute, with the below:

1. Get a domain from a website such as Namecheap. Eventually, the website will use this domain with https support. The domain name should match the name of the project, Costly Commute.
2. There should be a Postman API Schema with information about the API endpoints. The endpoints should be centered around the models used in Costly Commute. The technical report should contain a link to the schema documentation.
3. Host the website on the domain using various AWS services. AWS Amplify could be used for React or an S3 bucket could be used for a purely static website with only HTML/CSS. The pages on the website will be served through AWS.
4. Create the splash, model, and instance pages that will likely be only static. There should be at least 13 pages (every page except for the about page). The model pages should have the instances displayed in a grid or a table and the instance pages should have a lot of attributes and some media displayed.

5. Create the about page of the website, which contains information about the developers of the site. The page should show a picture and GitLab information about the repository used to develop the site. The GitLab API should be used to dynamically get this information.

Phase 2

Tasks

Pagination

On the backend side, the API returns data paginated, so the frontend does not have to slice the data itself. On the frontend side, implementing pagination was fairly simple because we are using Material UI. Material UI already has a package for pagination that we could install using npm and use to implement it. The Material UI pagination uses a button component in its implementation. To use the package, after loading the model instances data using the API call, we just had to specify the offset and the total amount of pages. Our pagination is showcased on the bottom of the city and doctor model pages. The path to our pagination implementation is `doctorsearch/frontend/src/component/pagination/pagination.js`.
For the specialties model, we used a table to display the data, and Material UI has a TablePagination package that made implementing pagination for tables very simple.

Database

We created a PostgreSQL database hosted on AWS RDS. Using the data sources described below, we first seeded our specialties table. Then, we seeded our cities table. Finally, we seeded our doctors table. During this, we added connections between the doctor, its city, and its specialty so that the API server would be able to easily get related instances of a data instance when querying the database. Before looping over the doctors to add them, we queried all of the cities and specialties and stored them as dicts mapping from name to the SQLAlchemy model object. While looping, the code found the name of the city that the doctor is in and the name of the specialty that the doctor practices from the BetterDoctor API response and got the related city object using the city name and the related specialty object from the specialty name. Then, the code added the doctor to the city's list of doctors and the specialty's list of doctors, which connected the instances.

Models

There are three models for the website. For our data, we scraped from the Teleport API (https://developers.teleport.org/api/), the GeoDB Cities API (http://geodb-cities-api.wirefreethought.com/), the BetterDoctor API (https://api.betterdoctor.com), the Google Maps Embed API (https://www.google.com/maps/embed/v1), and the Google Maps Static API (https://developers.google.com/maps/documentation/maps-static/intro).

1. Doctors
    a. **Description:** Doctors that can provide medical services. The model page contains a profile picture to identify the doctor and lists five attributes describing the doctor. Users will be able to sort this model by city, specialty, rating, name, and gender. Each instance page contains the list of attributes, a profile picture of the doctor, and a picture of the doctor's location on a map.
    b. **Attributes:**
        i. Name
        ii. City
        iii. State
        iv. Phone
        v. Specialty
        vi. Address
        vii. Insurance plans accepted
        viii. Rating
        ix. Biography
2. Cities
    a. **Description:** Cities in the United States. The model page contains a picture of the city and a list of five attributes that list statistics about the city. Users will be able to sort the cities by population, state, number of doctors within the city, specialties, and name. Each instance page of this model includes a picture of the city and a map of the city.
    b. **Attributes:**
        i. Name
        ii. State
        iii. Population
        iv. Time Zone
        v. Coordinates
        vi. Number of Doctors in the city
        vii. Number of Specialties
        viii. Elevation
3. Specialties
    a. **Description:** Different types of medical specialties practiced by doctors. The model page contains a representative picture of the specialty and a list of five attributes that describe the specialty. Users will be able to sort the specialties by city, number of doctors, name, and category. Each instance page of this model includes a representative picture of the specialty and a map of places in the United States where the specialty is practiced.
    b. **Attributes:**
        i. Name

ii. Category
iii. Number of Doctors
iv. Doctors practicing it
v. Cities it is in
vi. Description

Testing

We wrote unit tests for the backend, which can be found in `backend/tests.py`. These tests use the Python in built `unittest` package and cover various aspects of the code used to seed the database. We also wrote some unit tests for the API using Postman. After a request is sent to the API, the tests are run on the response to make sure that the API is working as expected.

As for the frontend, we used Mocha and Chai to write unit tests for various parts of the frontend. We also used Selenium and Python to write acceptance tests for the frontend, which can be found in `frontend/guitests.py`.

User Stories

Customer User Stories

We were tasked with the below for phase 2:

1. **Ratings on Doctor instances** - I noticed that some doctors do not have ratings, so in the cards for doctor instances, the ratings look like '/5'. Is there a way you could change it to provide a better default if no rating is provided? For example, 'Rating: Not rated yet' or 'Rating: */5'. Just something to indicate to users that the doctor has not been rated/no rating has been found.
   Time Estimate: 10 minutes
2. **Some pages of Doctor does not load.** - In the doctor tab, there are some pages that do not load such as page 4 and 10. Trying to press back on the browser does not fix the issue. We think it is because you are not doing null checks.
   Time Estimate: 10 minutes
3. **Rework of Doctors Instances** - I noticed going through your website that the instances certain doctors are a bit messed up. There is no image/map location for these instances, and instead, there is only the image placeholder icon with 'docimg/locimg'. If you could fix that so the proper image appears that would be awesome.
   Time Estimate: 2 hours
4. **Have a better description in About page** - You guys have done a fantastic job on your website! However, I feel like the about page doesn't represent what you can truly do with doctor search. The about page currently has the description, 'We help people find doctors around the United States.' Is there any way you can make the description a little bit more specific, and talk about the features that help a user find a doctor?

Time Estimate: 15 minutes

5. **Information in cities instance** - The information under the cities instance could use some more attention. I think that making the categories bold and right justifying the text might make it look a little cleaner. Also population is missing a :.
   Time Estimate: 30 minutes

User stories 1, 2, 3, and 5 were formatting fixes. For user stories 1 and 3, we wrote some functions to format data from the API to handle missing values, including rating and image. We also used the Google Maps Embed API to show a map of the city or the location of the doctor on the instance page. User story 4 was also simple as we just added some more information to the about page.

Developer User Stories

Our developers are Costly Commute and the link to their website is https://www.costlycommute.me. Here are the user stories that we gave them for phase 2:

1. **Add more instance pages for each model** - Collect data on many instance pages of each model. Each model should have data that has been retrieved using a RESTful API. The frontend server must use a RESTful API that is provided by the backend server.
2. **Pagination** - Once you have added more instance pages for each model, add pagination. Pagination should allow customers to easily navigate through the instance pages. The use of pagination should make the website seem more organized.
3. **Adding media to instance pages** - The instance pages should have more media. The instance pages for the commute and housing models only have one image per page. There should be a minimum of 2 types of media per instance page.
4. **Format the data displayed on instance pages** - As of now, the data on the instance pages do not have any units, so units should be added so the customers can use the information that is presented on these pages. Also, display the data in a format that is more readable for the customer. The maps under the Commute model are small and difficult to see.
5. **Connect the models to 2 other models** - Models not connected/linked to other models. All models should be interconnected to at least 2 other models. Instances should include links to an instance of each other model.

Phase 3

Tasks

Filtering

To implement filtering, we added dropdown elements to the model pages where users can select one of the options to filter the instances by. For example, on the doctors page users can filter doctors by state, gender, and title. When the user sets one of the filters, an event handler runs that sets the state of the page and updates the `filterQuery` JavaScript object in the state. For example, if the user filters by California, the `filterQuery` object would be `{state: 'CA'}`. Filters are additive, so choosing multiple filters updates `filterQuery` with multiple attributes.

> For example, selecting Female on the gender dropdown after selecting California on the state dropdown would result in a `filterQuery` of `{state: 'CA', gender: 'Female'}`.

This filterQuery object is passed into getCities(), getDoctors(), or getSpecialties() (found in `doctorsearch/frontend/src/datastore/DoctorSearchData/DoctorSearchData.js`) as a parameter depending on the model page that the user is on. Within these three methods, the code makes sure that the attributes in `filterQuery` are valid by checking if each attribute is a member of the list of valid attributes to filter by for the model. Then, a query string is constructed of the form *?q={"filters":[{name: `<attribute>`, op: 'eq', val: `<value>`}]}*. As part of the query parameter, a list of filters is passed in, and each filter is of the form {name: `<attribute>`, op: 'eq', val: `<value>`}.

> For example, `{state: 'CA', gender: 'Female'}` would turn into the query parameter ?q=`{"filters":[{"name": "state", "op": "eq", "val": "CA"}, {"name": "gender", "op": "eq", "val": "female"}]}`.

This query parameter string is appended to https://api.doctorsearch.me/api/`<model>`, where `<model>` is **city**, **doctor**, or **specialty**. In the backend, Flask Restless takes these filters and returns paginated results with the filters applied, which are then displayed on the page.

Searching

For searching, we first created a React component called `CssTextField`, which is a wrapper around the Material UI component `TextField` with extra styling. This component is used as the search bar on the model pages for model search and the search bar on the splash page for sitewide search. When a user enters a search query and hits enter, an event handler runs that opens a search page. Right before the search page renders, it makes an API call to the backend of the form https://api.doctorsearch.me/api/search?q=`<query>`. In the backend, there's a Flask route that handles the query. In the route, the code loops through all of the instances in the database and checks if the search query appears in any of the attributes in each instance. Then, it returns the instances of each model that match the search query as the response. Once the search page gets the response, it displays the name and some information about the instance for each instance in the response. When rendering the page, we use a package from npm called `react-highlighter`, which handles highlighting the exact text that matches the search query.

Sorting

We implemented sorting very similarly to filtering. We added dropdown elements to the model pages where the user can select Ascending or Descending for the attribute to sort by. When a user sets one of the sorts, an event handler updates the state of the page in the `sortQuery` JavaScript object. This object is passed to one of the methods to get model instances, which checks if the attributes passed in are valid to sort by and then constructs a query string of the form *?q={"order_by":[{field: `<attribute>`, direction: `<asc|desc>`}]}*.

> For example, if a user sorts by population ascending on the cities page, the sortQuery object `{population: 'asc'}` would be constructed and then passed into `getCities()`. Then, it would turn into the query parameter `?q={"order_by":[{"field":"population", "direction":"asc"}]}` inside the method.

This query parameter string is appended to the https://api.doctorsearch.me/api/`<model>` and on the backend, Flask Restless handles the sorting and returns paginated results with the sorting applied, which are then displayed on the page.

User Stories

Customer User Stories

We were tasked with the below for phase 3:

1. **Create an Icon** - I think having an Icon would help Doctor Search as a website! This way, you could have something as the favicon instead of a random Asian (presumably) Doctor. It also might help in re-designing the splash page!
   Time Estimate: 30 minutes
2. **Make each row in table of specialties clickable** - Currently, you have the specialty as the only part clickable in the table to redirect me to the specific instance. I think it would be a good idea if you made the whole row clickable to redirect. One reason is to be consistent since your other pages do with their respective card components.
   Time Estimate: 1 hour

3. **Add filtering capabilities to get more relevant searches** - Right now it's hard to find relevant doctors. We don't want to be clicking through all the pages of data to find our doctors and want to have a better experience finding relevant results. Maybe returning results in a list/table fashion would make it easier to look through results, but up to y'all if you want to keep the same block display for search results.
   Time Estimate: 3 hours
4. **Make developer blocks on the about page same size** - So I noticed that the about page is a little asymmetrical. Rohan's block is a bit bigger than the other group members'. I think this is caused by the picture, so hopefully cropping that will fix this issue!
   Time Estimate: 1 hour
5. **Create a search bar for easy searching** - Having a search bar would make it really easy for us to find specific instances of doctor, city, and specialization. Maybe have each page have a different default text for the search bar. For example, the empty search bar for the city page could say something like "Search Cities" and the specialization page would have the empty search "Search Specializations".
   Time Estimate: 5 hours

User stories 1, 2, and 4 were minor visual fixes. For user story 1, we made an icon using https://gauger.io/fonticon/ and used it on our splash page and as our favicon. For user story 2, we moved the `onClick` function that changed the page from the specialties page to the specialty instance page from the table cell containing the name to the table row. For user story 4, we set a minimum height on the `UserBio` React component and cropped the pictures so that the developer blocks would be the same size. User stories 3 and 5 required more work and the implementation of filtering and searching is discussed in their respective sections above.

Developer User Stories

Our developers are Costly Commute and the link to their website is https://www.costlycommute.me. Here are the user stories that we gave them for phase 3:

1. **Add a description of your website to the About page** - You guys came up with a great idea for the website as it's something that I think many people would use. However, there's currently no description on the About page. It would be nice if you guys added something talking about how exactly the website works and various features.
2. **Page numbers are incorrect on the model pages** - On the model pages, the page numbers at the top say '1 2 3 4' for the first 4 instances. Clicking next takes you to the next 4 instances, which are '5 6 7 8'. Instead, one page should have multiple instances under it, so clicking next should only increment the page by 1, not 4.
3. **Make search form on left specific to each model** - On the left of each of the model pages, the search form has 3 tabs, 1 for each model. However, it seems unnecessary to have 3 tabs as on the cities page, you don't need to search for commutes or homes, and vice versa. Therefore, on each model page, there only needs to be a search form for the

current model, without the 3 tabs. Also, there's a lot of unnecessary whitespace above the textboxes in the search form.

4. **Showing more information about the instance when clicking Select seems unnecessary** - When you click Select on an instance, more information about the instance shows up on the left, along with an image. This seems unnecessary as you could just show 2 or 3 more points of information within the card and all the information on the instance page when the user clicks Read More. This would allow you guys to worry about less functionality/code to maintain. Feel free to push back on this story if you guys really want to keep the feature. I would enjoy a discussion 😃
5. **Fix filtering and searching and add sorting of instances on model pages** - Currently, the filtering and searching doesn't seem to be working correctly. For example, when I enter New York as the city in the form on the cities page and hit 'Submit Query', New York doesn't show up for me. Also, there should be a way to sort instances based on some of their attributes.

RESTful API

Postman API Documentation: https://documenter.getpostman.com/view/9000368/SVtbPkAt

The Postman API documentation is split into four sections, **doctors**, **cities**, **specialties**, and **search**. The first three sections refer to the three models described above. Each model has two endpoints that correspond to it. Using the model **doctors** as an example, we have:

- `/api/doctor`
    - This endpoint allows users to get a list of doctors and filter them based on the optional parameters that are described in the documentation.
- `/api/doctor/{id}`
    - This endpoint allows users to get a specific doctor based on the ID that uniquely represents it.

The other two models each have two endpoints that work in the same way, which will eventually allow the frontend to dynamically pull data to display.
The fourth section, **search**, has one endpoint:

- `/api/search?q={query}`
    - This endpoint allows users to search the entire database for model instances that have attributes containing the search query.

The documentation was generated using the Postman desktop client. We sent requests to our API endpoints and wrote tests for them to make sure that the API works as expected.

Tools

- **GitLab** - We used GitLab to store the source code to DoctorSearch and. We also used it to version control our code and track issues.
- **Postman** - We used Postman to send requests to our API server, write tests to make sure our API is working as expected, and generate documentation.
- **Material-UI** - We used Material-UI as our CSS Framework to make sure our website is responsive. We constructed the navigation bar and grid of model cards using CSS components from the framework. Our pagination was also built using Pagination and TablePagination components from Material-UI.
- **Slack** - We used Slack for communicating within the team.
- **React** - We used React as our frontend framework to build the website, handle the routes for the different pages, and create UI components that could be reused across pages.
- **React-Router** - We used React-Router to build routing for the website so that users could go between the home page, about page, model pages, and instance pages without doing a full page reload.
- **Flask-SQLAlchemy** - We used Flask-SQLAlchemy to define our models for City, Doctor, and Specialty. For each model, we defined its attributes and its relationship to the other models.
- **Flask** - We used Flask to build our backend API server from our data models. Thanks to Flask Restless, we didn't have to define the API routes to get model instances ourselves. We manually wrote a route for searching using Flask.
- **Flask Restless** - We used Flask Restless to generate API routes based on the models automatically. It generated a GET request for all data instances with pagination and a GET request for a data instance by ID for each model and connected the models based on the defined relationships.
- **pgAdmin 4** - We used pgAdmin 4 to inspect our PostgreSQL database.
- **Black** - We used Black to format our backend Python code.
- **Prettier** - We used Prettier to format our front-end JavaScript code.
- **ESLint** - We used ESLint to lint our front-end JavaScript code and help catch bugs.

Hosting

- **Namecheap** - We used Namecheap to get our domain, doctorsearch.me.
- **AWS Route 53** - We used AWS Route 53 as our DNS resolver for our website. To achieve this, we created a new hosted zone in the AWS Route 53 console, which automatically generated nameservers. We then entered the nameservers in the Namecheap domain settings so that AWS Route 53 would handle DNS resolution instead of Namecheap.

- **AWS Amplify** - We used AWS Amplify to host our React application. We made a new project in the AWS Amplify console, which we connected to our GitLab repository. We then wrote a build script that AWS Amplify runs every time a new commit is pushed to the repository, which runs `npm run build` in the `frontend` directory to update the website. We added our AWS Route 53 managed domain to the project, which automatically handled SSL certification.
- **AWS Elastic Beanstalk** - We used AWS Elastic Beanstalk to host our backend API server. Using the Elastic Beanstalk CLI, we created a new project and environment and deployed our Flask application (in the `backend/` directory) to our environment.
- **AWS RDS** - We used AWS RDS to host our PostgreSQL database.