StudySpots.me

Authors: Bianca Alvarado, Joanne Chen, Vincent Chen, Ami Iyer, Jake Medina

Motivation

As college students, it can be hard to balance the responsibilities of studying for classes with the desire to explore the city and enjoy the time at college. On top of this, there are other circumstances and new environments and experiences to adapt to that make it more difficult to balance everything in life.
Our goal was to help college students balance these responsibilities. We aim to provide a variety of different areas that can fit the circumstances that a college student might need at any given time to work at (i.e. a quiet place, a place that opens for long hours, a place that has cheap food, etc.) and that is around different locations to encourage students to explore more places around where they are.

User Stories

Received From Customer

User Story: Filter Study Spots by Closing Time

> *I am a busy university student who often studies late in the evening. I would like to be able to quickly find study spots near me that open until late. Implement a feature that allows the user to filter coffee shops and/or libraries by closing time.*

The hours that the place was open was something we wanted to include because we know it's a useful attribute to consider when choosing a location to work at. Adding a filter to sort the place by their hours is a good idea for convenience and just to make sure the place is open! We will implement this as something that can be sorted by in phase 3.
Hello, we have added the ability to filter by closing time for the coffee shops page. There is a new filter box labeled 'Open until:' that narrows down coffee shops to the ones that are open at or before your selected time for the current day. If you select the 'Show all' option, it will show all coffee shops, regardless if they are closed for the day or the hours information is unavailable

User Story: Recommend Coffee Shops Closest to My Location

> *I am a busy university student who likes to study at coffee shops. I would like to quickly retrieve a list of coffee shops closest to my locations. Implement a feature that allows the user to obtain a list of the 3 closest coffee shops along with their description (hours, rating).*

This is a really good idea and one of the initial reasons that led to us developing this idea. We'll have to look more into the feasibility (legally and technically) of asking a user for their location, but if everything checks out, we'll add this into the sorting/filter logic in phase 3. If we run into issues, we'll definitely still look into using the location to sort but in a different manner.
In phase 3 we are creating a zip code filter field for each model page. You can enter your zip code here

and see all coffee shops in your area. You can also do this with libraries and universities as well. When this feature is pushed to the develop branch, this issue will be closed.
Estimated time: 4-5 hours
Actual time: 5-6 hours (Time includes overall time spent to implement exact features since this issue was something we wanted to filter with).

User Story: Custom User Locations

> *As campuses are quite large, it would be useful to be able to input your own location and search from there.*

This is a really good idea and one of the initial problems that we faced that led us into thinking about study spots as a possible idea for the project. We'll have to look more into the feasibility (legally and technically) of asking a user for their location, but if everything checks out, we'll add this into the sorting/filtering logic in phase 3. If we run into issues, we'll definitely still look into using the location of the places themselves to sort, but in a different manner.
In phase 3 we are implementing a filter on every model page where you can type in your zip code and see all of the results nearby. This keeps the user's location private and avoids cumbersome dragging and clicking on a map. When this feature is pushed to the develop branch, this issue will be closed.
Estimated time: 4-5 hours
Actual time: 5-6 hours (Time includes overall time spent to implement exact features since this issue was something we wanted to filter with).

User Story: Estimated Travel Time to Location

> *It would be nice to have an estimate for how long it takes to get to various locations like in google maps; i.e. walking, biking, by car, etc. Just having walking should be sufficient though.*

This is a really cool attribute that we didn't think about including. We can include this extra information about transportation between locations in the instance page once we begin adding the connections between the different models. We'll look into adding these connections and extra information in phase 3 since it seems to rely on information about a user's location that will be possibly added in phase 3 with the filtering/sorting logic.
We were able to implement this issue this phase! We have implemented this feature under the 'Map' section. There is now a button that will request your location through the browser and calculate the walking travel time in Minutes+Seconds to the instance.
Estimated Time: 2-3 hours
Actual Time: 3-4 hours to research APIs that we needed to invoke to extract the information and to take in the user's location as well as implementing this feature

User Story: Study Spots That Do Not Require Purchases

> *I am a university student who needs a place to study regularly. I would like to be able to find study spots that do not require purchases. Please implement a feature that allows the user to find study spots that do not require purchases.*

This is definitely a sentiment that a lot of college students share. We will have to look more into whether or not there is any way to identify which coffee shops require purchases since that knowledge is more

likely to be informal knowledge based on word-of-mouth of which places are more relaxed in their purchase requirements, so it might be out of scope. Nonetheless, we'll look further into this in phase 3 when we're starting to add more sorting and filtering logic.
Unfortunately, this type of information is not provided by any of our model source APIs. Neither Yelp or Google Maps' API endpoint will tell you if you have to purchase something or not. Therefore, we cannot implement this feature.
Adding on, we did our best to try to provide a way for you to identify cheap places that you could still visit; we added a filter/sort based on the price of the place so you can select only the cheap coffee shops as places that you visit. Additionally, the libraries should always be free for you to visit and study there. Hopefully, these are both solutions that help mitigate some of the costs of studying at outside places.

User Story: Allow Easy Copying of Study Spot Location

> *Hello. When studying, it's often nice to do it with other people, but communicating the location to them can be a real pain, especially when you don't already know where the place is. It's a lot of work to manually type addresses and place names etc. I would like a way to easily be able to copy the relevant information of a study spot so I can paste it into a messaging app.*

This is a good idea, especially since it might not be easy to select the text to copy for those who are trying to access the website on their phone. We'll try our best to implement this feature this phase, but since this request was given close to the deadline, if we cannot meet the deadline we will implement this feature first thing in phase 3.

We were able to implement this feature during this phase! We added a button that you can easily click and it will automatically save the address and website of the place to your clipboard. Hopefully, this makes it easier to copy information and share that information with your friends. Adding this feature took roughly 30 minutes.
Estimated Time: 45 minutes
Actual Time: 30-40 minutes to provide a button that copied all of the information to the user's clipboard.

User Story: Allow Jumping To Various Pages in the Search

> *When using sites containing searches with many pages, it can be very inconvenient to lose your spot and be unable to access it again without manually going next page until you reach where you were. I would like a way to jump multiple pages at once, so I can get back to where I was easier. I would also like a way to go to the end of searches, if I want to see results going backwards instead.*

This is definitely a feature we will try to support in our pagination logic! We will try our best to support this in phase 2 while we build our pagination logic, but if not and we are only able to support skipping a single page for the pagination logic in the meantime, we will definitely revisit this in phase 3.
We were able to implement this during this phase! We added an array of page numbers at the bottom of the model pages such that the first few numbers and the last few numbers are displayed in the array. This way, you don't have to constantly click the next page to reach the last page and can immediately jump to the last page.
Estimated Time: 2 hours

Actual Time: 4 hours to implement all of the Pagination logic and provide an array of numbers at the bottom that the user can jump between.

User Story: Display Today's Weather at the Location

> *The weather is really prone to changing, and I sometimes go outside, only to immediately return to grab a jacket because I wasn't expecting how cold it was. It would be nice to be able to see the weather at a location, or at least just the city. This would save me time running back and forth, especially when I'm in a hurry to be on time.*

This is an interesting idea. We won't be able to add this information to the backend database since that is static and the weather will be constantly changing over time, so we'll have to look into adding the extra information through dynamic API calls in the frontend. This might be out of the scope of the project, but we will try to support this feature in phase 3.
We were able to implement it this phase! You can see the current weather conditions at the top of every instance page, just below the name of the instance.
Estimated time: 1-2 hours
Actual time: 2 hours to research into other APIs we could use and then extract the data and display that on all the instance pages.

User Story: Display the Time Zone of the Opening/Closing Times

> *The internet has made everything feel closer together, but it's also cause some problems regarding geographic locations. It's annoying when the time zone of a location is ambiguous or needs to be inferred. I would like to have the time zone be displayed with the opening/closing times of the libraries and coffee shops.*

You bring up a good point that we overlooked -- we forgot about the scope that our website might be visited, and made the assumption that the audience would mostly be UT Austin students and thus would be in Central Time. In the meantime, you can assume all the time zones are in Central Time, but we will look into adding more processing logic to display multiple time zones in phase 3.
We were able to implement a simple solution to this issue this phase! We added clarifying labels to the hours to indicate that the time is in Central Time. We were contemplating displaying the information in the time zone that the user is in, but this adds some complication since the hours are stored in the database statically as a formatted string of hours in Central Time Zone, so there is no easy way to be able to access the user's timezone, parse the string for the hours, and convert the hours relative between Central Time and the user's time zone. Hopefully this solution still brings some more clarity when you're viewing it at least!
Estimated Time: 10 minutes
Actual Time: 10 minutes to implement and brainstorm possible solutions to the issue

User Story: Coffee Shop Images

> *I am a university student who enjoys studying at coffee shops. I would like to be able to get a good idea of what the coffee shops look like before deciding I want to study there. I would like to see more images of the coffee shops, especially the interior.*

We will look into adding more photos and having a slideshow of the photos displayed though there might be some limitations because of the amount of information that the APIs return. Given the upcoming deadline for phase 2, we probably will not be able to implement this in time, but we will look into adding this for phase 3!
We were able to implement this issue this phase! We now have a slideshow of three photos displayed on each coffee shop instance page. Unfortunately, since the Yelp API only provides us a maximum of three photos (or less if there isn't enough information), we aren't able to provide more images in the slideshow. Hopefully these three are still enough to give you more of a sense of what the place looks like before you decide to go there!
Estimated Time: 1-2 hours
Actual Time: 30-45 minutes because we actually ended up having all of the information we needed in our JSON files, so we didn't have to modify the backend significantly except to extract that information, and we had to add a Carousel component for our images.

User Story: Improve the formatting of the model pages

> *I see that all the information is on the model page, along with a nice button that lets you copy information to the clipboard. I do want the layout to look nicer though; for example, choosing new fonts, a good color palette, or even just improving the layout of text. This would make the model page more readable and just nicer to look at.*

We will definitely look into trying to make the instance pages look more neat and nicely formatted by having more clear labels and separation of the information we provided. We initially created the instance page template the day the first phase of the project was due and never had the chance to go back and fix the template to make it more uniform and comprehensible.
We were able to implement this issue this phase! We changed the layout of the instance pages by grouping together related information (rather than having the maps be next to information about the place's phone number) to avoid the confusion of providing too much information at a time. We also changed the layout to be more of a vertical-scroll with the grouped information on top of one another to help focus in on key ideas. We also switched to using a more casual font to give a more friendly rather than formal vibe.
Estimated Time: 2-3 hours
Actual Time: 3.5 hours to remove unnecessary style files and classes and to implement new ones that can be reused across all instance pages.

User Story: Provide sources for copyrighted images and data sources

> *This is an important part of using other people's intellectual property, even under fair use. Given the public nature of the website and the significant value of images in the website's design, the images should be cited. For model images, providing the API is sufficient, as it would be excessively burdensome to manually cite all of them.*

Thanks for pointing this out! It's definitely important in this digital era to make sure to credit the correct person. I don't know if we will have all of the information about the image to give it proper citations (like the image author, image creation date, etc.) because the API only provides links, but we will definitely include the links on our instance page to be able to provide at least some form of citation. We will definitely be implementing this in this phase!

We were able to implement this issue this phase! We made a simple change to add a caption to all of the images displayed on the website that states the URL that the image is taken from.
This change only took 10 minutes to implement because we already had all the information at our disposal to be able to quickly add another component to give the pictures a caption with its url.
Estimated Time: 10 minutes
Actual Time: 10 minutes because we had all the information available and simply had to add additional components to display it

User Story: Add a dark mode

> *This is an important part of using other people's intellectual property, even under fair use. Given the public nature of the website and the significant value of images in the website's design, the images should be cited. For model images, providing the API is sufficient, as it would be excessively burdensome to manually cite all of them.*

This is definitely a cool idea! Bright screens definitely do put a strain on my eyes, so the dark mode option would be a cool toggle to be able to implement. We will look further into this and aim to have this finished by the end of this phase.

We were able to implement this issue this phase! We have a toggle button on each page for easy access, but changing it should persist across all pages until you change it again. The light mode will display everything in a lighter shade of brown (to still try to avoid hurting your eyes even in light mode!) and the dark mode will display everything in shades of dark blue.
Estimated Time: 1-2 hours
Actual Time: 2-3 hours because of having to add CSS components, research how to switch CSS styles based on the mode and how to persist the mode across pages, and changing all of the components to use this style

User Story: Identify which category the links on model pages fall under

> *It gets a bit confusing identifying which link is a coffee shop, library or university when looking at a model page, as often those share the same words similar words. For example: is the Bioinformatics Research Center a university or library? Clicking on the link reveals it's a library, but it isn't immediately obvious. Having a header or label separating these links would help a lot with readability.*

This is a good point! It might not be intuitive in all of the cases from the name what type of place it is, and it would be a waste of the user's time to have to check out each of the links if they were exclusively only looking for libraries nearby their university. We'll have this added by the end of this phase!
We were able to implement this issue this phase! We made sure to add a header to indicate which model page the links are for. To make sure that it was extremely clear, we also rearranged the layout of the component that found the nearby instance pages by returning them as boxes that had a list of the places nearby rather than just a plaintext list of the places. This way, you have a box with a header indicating what model page these links are and the list of links themselves.
Estimated Time: 15 minutes

Actual Time: 30 minutes to implement the feature that was requested by adding a title to our component but also some time spent to change the overall structure of the component

User Story: Have some way of identifying what type of model page you are currently looking at

> *At the moment you essentially have to deduce which type of model you are currently looking at from context clues. It would be nice to have some visual indicator of what model category you are looking at when on a model card's page. This could be with an icon (coffee cup, book, building), a highlight on the navbar, or even just explicitly noting what type of model it is via text.*

This is a simple and good idea to help bring even more clarity to the user regarding the instance page they are looking at! This can especially be helpful in cases where they are following links to other instance pages because it can help reconfirm to them what link they clicked on and what places they are looking for. We will have this implemented by the end of this phase!
We were able to implement this issue this phase! We added a very simple design by having a divider between the name of the place and the rest of the information, and inside that divider, there is an icon/emoji that indicates what model type it is (e.g. a coffee cup for the Coffee Shop models, a book for the Library models, and a graduation cap for the University models). We thought this would be a simple and clean way to still remind you what type of model it is without being too intrusive for those who already know what model pages they are on.
Estimated time: 20-30 minutes
Actual time: 15-20 minutes to brainstorm the idea and add the emoji and extra Divider component to each model page.

Given to Developer

User Story 1: I'm a user that isn't too familiar with the rules of soccer. I did a brief search and was surprised to find out that soccer teams can choose how many members they want to include on their roster even though only 11 players can be on the field at a given time. I'm curious how many players most teams usually have on their rosters and if there are any correlations between more successful teams and the size of those teams. Thus, I think it would be an interesting attribute to include (and filter by) in the instance page for the soccer teams.

User Story 2: I am a user that tends to always have multiple windows open when working and looking up information. As a result, I usually have to resize my different windows in order to fit them all on my one laptop screen. I really like using your website to lookup information, but when I resize the window to be able to have another window to type the information I learn from your website, the instance cards on the model pages ends up getting very squished that it is sometimes harder to read and focus on. I would like there to be an option to change how many instances are displayed per row so that the cards don't get too squished and narrow so that it becomes harder to read when the window is resized.

User Story 3: I am a user who has a slow computer and sometimes all of the pictures on the website does not load fully and I can only really see the text. Additionally, I have a hard time understanding what the different team logos are trying to show/be. Therefore, I would like if there could be more information

added to the instance page itself that details what the team's mascot is and any information about that mascot (like their name).

User Story 4: I'm not the best at geography, so I can't really envision in my mind where all the countries that you have listed on your website are located globally. I do know where all the general areas are (Europe, North America, etc.), so I would like if there was some additional attribute included that just stated what general area the country was located. I would also like if there was a filterable property based on that since it would provide valuable information on where soccer is generally most played.

User Story 5: I don't know much about soccer, but I have some knowledge of how other sports games work. Depending on how far a team might advance in championships/playoffs, different teams can play different amounts of games. Thus, I was wondering if you could instead also add an attribute for the general team win percentage to take into account both the number of games a team played and how many of those they won (rather than only showing the teams that have played more games and thus won more games than other teams). Additionally, it would be nice to sort by this.

User Story 6: On the players/teams instance page, try to resize the logo and format it so that it looks less blurry/more organized with the information. The pages are also not very interactive. Maybe you could embed a video of the player to have something more dynamic on the page than just two photos, especially since the photos between the team and player repeat.

User Story 7: For both players and teams, see if you can scrape more information describing the team to add more info overall. Perhaps wikipedia's intro paragraph would be a good inclusion. I think linking the social media of the players would also help make the site more engaging.

User Story 8: Thinking about sorting/filtering, all or most of the data on your cards seems to be numerical. Maybe try moving a categorical one there too i.e. the country a player is from to offer more than just stats to filter by. Moreover, thinking about what people want to learn about players will most likely want to filter by nationality, team, age, and maybe some skill-based stats/their earnings.

User Story 9: The stats included countries page is a little confusing. Maybe organizing the time-zone table would help, since the purpose of that is a little unclear. It would also be cool to have more soccer related info on the countries page since that would be the focus of the website.

User Story 10: The article linked in the countries page is also a little random in the scope of the website. If possible, narrow it down to some sports article. I think containerizing these elements would also help organize the information in a more readable manner.

User Story 11: I'm a user that tends to open up multiple things at once to be able to have access to all of the information that I need at a time. With your website, it seems that all of the buttons will just replace the current website with the new website that the button was linked to. As a result, I can't easily see multiple tabs at once, so I think it would be a nice feature to have all the buttons open in a new tab by default.

User Story 12: I'm a user that would like to engage in some user interactivity with the website and to be able to continue seeing and reading information while I might be interacting with another component of the website. With your website, it seems that you could allow for some user interactivity directly in your website by embedding the Google Maps directly in the instance page rather than redirecting to an external link, and I could still be able to read the information in the instance page while looking at the map. Is it possible to be able to directly embed the maps of the countries in the instance page rather than redirect to an external website?

User Story 13: I'm a user that really learns best with visual inputs rather than primarily textual inputs. A lot of the information that your website provides seems to be statistical and numerical information. I think it would be nice to include a slideshow of more images (rather than just a single image representing the instance) that the user can look through while they are reading the information, which would allow for the user to draw more visual connections between the information provided and the images. Notably, for countries, I think it would be nice in addition to also have images of the countries themselves rather than just the flags to provide a fully encompassing picture of the instance.

User Story 14: I'm a user that understands the content well if it is formatted well. In your website, I really like how you had whole cards to link to the other instances in the model, but I was a bit confused initially because it was inline with the rest of the information for that specific instance. In other words, I was confused initially whether it was supposed to be supplemental information for that instance or separate information about linking. To make it more distinct, I think it would help to have the instance pages formatted more intuitively to keep different aspects of it separate (e.g. having the links to the other instance pages on a separate line at the bottom whereas the information for the instances are more towards the top).

User Story 15: I'm a user that likes to be able to understand all the parts of a component. The website right now seems to mostly provide connections between an athlete to what team and country they are from and a team to what country and league they are in. I think it would also be nice to include links (maybe doesn't need to be entire cards since there will be more of them) between an athlete to other athletes that are on the same team as them and between a team to the athletes that are on the team. This way, I can fully know regardless of what instance page I'm on the entire team make up and know where to look for more information about that team.

RESTful API

Link to our API documentation: https://documenter.getpostman.com/view/23653833/2s83tGoBu1
The endpoints in the form of “api.studyspots.me/<model name>” such as “api.studyspots.me/coffeeshops” by default return a list of the first 10 instances of that model. It will return all of the field information. To change the number of instances returned and which instances are returned, the parameters “page” and “per_page” can be supplied with numbers.
The endpoints in the form of “api.studyspots.me/<model name>/<id>” will return all the fields related to the instance page of that model with that id.

Models

Universities

There will be roughly 200-300 expected instances for the Universities model.
Each instance will have the following sortable and searchable attributes: state, city, enrollment, names (alphabetical), size of the school, average tuition cost, year founded, type of college (public, private, community).
In addition to these, this instance will also have the following additional searchable attributes: phone, website, distance to nearest city, distance from me, zip code.
The different media types that will be on each instance page are the location of the university on a map, the link to the website, images of the university, and any videos that college might have.

Libraries

There will be roughly 500-600 expected instances for the Libraries model.
Each instance will have the following sortable and searchable attributes: ratings, location, business of the library, name (alphabetical), location, number of reviews, and hours.
In addition to these, this instance will also have the following additional searchable attributes: phone, website, amenities, distance from me, and size.
The different media types that will be on each instance page are the location of the library on a map, pictures of the library, library description, website link.

Coffee Shops

There will be roughly 500-600 expected instances for the Universities model.
Each instance will have the following sortable and searchable attributes: ratings, location, business of the place, name (alphabetical), number of reviews, prices, hours, local/chain.
In addition to these, this instance will also have the following additional searchable attributes: phone, website, wifi, amenities, and distance from me.
The different media types that will be on each instance page are the location of the coffee shop on a map, ratings, pictures of the library, website links, and library descriptions.

Connections

All of these models will mainly be connected to one another based on proximity in location.

Tools

Software

We used the following software to develop our website:

- **React** is an open-source frontend JavaScript library for development of UI components.
    - We used this to create the overall website UI design and structure.
- **React Router** is a library for routing in React, enabling certain behaviors like navigating among components and changing the browser URL.

    - We used this to create connections and allow navigation between the different pages.
- **React Bootstrap** is a library that has some preconfigured components developed by React.
    - We used this to add some components that it already had to our website.
- **Material UI** is a library that has some preconfigured components developed by Google.
    - We used this to add some components that it already had to our website, namely the components for the searching/sorting/filtering.
- **AWS Amplify** handles the entire solution for frontend developers to build, ship, and deploy applications on AWS.
    - We used this to host our website.
- **Docker** packages all of the software needed for development (specified libraries, tools, etc.) into containers that can quickly be run and deployed.
    - We used this to create and synchronize the environment for development by having it include all the necessary installations.
- **Gitlab** is an open source code repository that helps facilitate continuous development and collaboration with other developers.
    - We used this for source control, collaboration with each other, and to keep track of issues that we and our customers come across.
- **Elastic Beanstalk** handles hosting backend servers.
    - We used this in order to host our backend server and be able to handle requests.
- **PostgreSQL** is a database and supports searching and filtering.
    - We used this in order to store all the information that we would need for our model instance pages. We also used this to be able to search for specific instance page information to return to our endpoints.
- **Flask** and **Marshmallow** provides routing between different pages and creates schemas to format the information to be returned from the database.
    - We used this to create our endpoints for the backend server and to format the information that we wanted to return.
- **AWS RDS** is a fully managed, open-source cloud database that allows you to easily operate and scale your relational databases.
    - We used this in order to store our databases.

APIs

We used the following APIs to develop our website:

- **GitLab API**: https://docs.gitlab.com/ee/api/
    - Information from this API was pulled to find out the number of commits and issues for each member in order to dynamically populate the About page.
- **Google Maps API:** https://developers.google.com/maps
    - Information from this API was pulled to find out more information about libraries. Generally, this API provides further information on all of our models.
- **Yelp API:** https://www.yelp.com/developers
    - Information from this API was pulled to find out more information about coffee shops.
- **College Scorecard API:** https://collegescorecard.ed.gov/data/
    - Information from this API was pulled to find out more information about different universities.

- **Wikipedia API:** https://www.mediawiki.org/wiki/API:Main_page
    - Information from this API was pulled to add more information about the descriptions of universities.
- **SerpAPI**: https://serpapi.com/
    - Information from this API was pulled to add a photo of each of the universities in order to have enough media for universities.
- **Open Meteo API**: https://open-meteo.com/en
    - Information from this API was pulled to obtain the weather for a given location in order to address a user story.
- **Routes API**: https://routes.googleapis.com
    - Information from this API was pulled to provide travel time estimation in order to address a user story.

Hosting

We registered the domain name *studyspots.me* on NameCheap and hosted the React website using AWS Amplify. AWS Amplify is connected to our GitLab develop and main branch. It build sthe React website in the frontend folder and hosts the web application at that domain name. AWS Amplify connects to the domain name through the SSL/TLS certificate, resulting in the overall website being https://studyspots.me. The develop branch is associated with develop.studyspots.me and the main branch is associated with the domain name itself.
For the backend server, we hosted it using AWS Elastic Beanstalk. This created an EC2 Instance that is utilizing Docker to run the Flask app continuously on the instance. Namecheap then redirects the domain name api.studyspots.me to this instance to be able to service endpoints via api.studyspots.me.

Design

Overview of Implementation

Phase 1

For phase 1 of the project, the website is structured into the different pages: About, Splash, Universities, Libraries, and Coffee Shops.
The Splash page and About page were mostly done separately since they had distinct layouts compared to the model pages. Thus, both of these pages were manually created by just structuring the different components on the page and providing any links to other pages if necessary.
The About page also had the added complexity of having to call the GitLab API. Thus, it first tries to call the GitLab API to receive all the information it needs (number of commits, number of issues, etc.) and then after that fetching is finished, the page is rendered.
The Universities, Libraries, and Coffee Shops pages were all similar in structure but would just contain different information. Thus, for these pages, a common component was created to make a model page template that can then be reused for each of the different pages, just filling it in with different information.

Similarly, on each model page, there are also instance pages on each model page that will also be similar in structure. Thus, a generic instance page template was created for the individual instances to then be able to fill out the specific information but with the generalized template.

Phase 2

In order to first populate our databases, we had to scrape all of the APIs for information that we needed. As this information was queried, it was stored into JSON files in order to statically have the information available for future recreations of the database to avoid having to constantly query the APIs in order to fill in our databases.

For the University, we queried our main college scorecard API information for the bulk of our information and to search for all of the universities in the US. We also had to do additional querying to the SerpAPI and Wikipedia API since the original API lacked additional media and descriptions to make the experience as rich as possible.

With the university information, we then had a basepoint of what areas we wanted to search since our next two APIs, Google Maps and Yelp, required searching in a specific area. We then parsed the University information for zip codes and locations to provide us a general area to start our Google Maps and Yelp API searches.

After all of these calls were done, they were stored in separate JSON files for easy access for the database creation.

The database was created using PostgreSQL and AWS RDS. For each model, we created a schema with the fields that we wanted to have for that model (such as id, name, location, etc.). We then iterated through our JSON files to extract all of this field information, create a new instance of the schema, and then add it to the database, which was being stored via AWS RDS. PostgreSQL then allowed for easy access and filtering of these items in the database for returning in our backend API endpoints, which was set up through Flask and AWS Elastic Beanstalk.

We are able to look through pages of the data by implementing pagination on both the frontend and backend. Our backend API takes *page* and *per_page* parameters and paginates our large datasets with Flask-SQLAlchemy so that, by default, 10 results are returned per page, and you can view whatever page you like with the page parameter. We then used the frontend to create a page selector using React-Bootstrap. This page selector can be found at the bottom of each of our model pages, and it contains logic which will call the API for the page that the user selects and update the page with the new data.

Phase 3

For all of the sorting/filtering/searching features that were added in this phase, we used the searchParams from the react-router-dom. With this, we could directly update the URL of the page that we were on, so for all of our sorting/filtering/searching features, interacting with the frontend component would simply add the corresponding query in the URL of the page.

This way, we could directly take from the URL of the page (via the searchParams) whatever queries and information we then needed to pass back to the backend API. Thus, the frontend components just needed to keep track of a change and once it notices a change, it will add the query to the URL of the website. This state change of the searchParams variable will then trigger a useEffect hook that will re-render the page to display the new information obtained from the API.

Searching

For the searching portion of the project, we used Material UI to add a TextField component to the model pages. The user could then type into the TextField, and anything the user typed was monitored with a variable in order to have the user's input persist in the text box. Once the user hits enter, this changes the searchParams as stated previously and re-render the screen to reflect the entries with that search criteria. Once the results of the API was returned, we used react-highlighter for each field that we were going to display on the model pages in order to highlight any words in the fields that matched any of the words in the search query that the user entered.

Filtering

In order to implement filtering, the API was modified to use SQL-Alchemy to query the database based on columns that we deemed as filterable. We created two different classes of filters, exact and range filters. Exact filters were things like state, city, and zip code that can be easily queried using the .filter_by() command in SQL-Alchemy. Range filters were those that fell between two integer values, for example, a coffee shop's rating or a university's average SAT score. A user can pass in two integer values, a min and max, for a column that is deemed range-filterable and the API will return any results in-between those values. SQL-Alchemy has the ability to combine queries, so many different filters can be called simultaneously, allowing for fine-tuning of parameters and detailed results. On the frontend, we used the MUI library to create responsive dropdowns and sliders that allow the user to easily filter data. Dropdowns are used to select state, city, and zip code and provide suggestions as you type using one of our API endpoints. The sliders allow you to filter by numerical fields applicable to the selected model, and dynamically call the backend API when the ends of the sliders are dragged to a new value.

Sorting

Similar to filtering and searching, the frontend made heavy use of MUI components, particularly Select and MenuItem, were used to create the drop down for picking what features to sort by (a set of sortable features were defined in UniversityOptions, CoffeeShopOptions, and LibraryOptions). Two main functions are used for sorting, one that handles a change in the menu item selected (handleSortChange) and one that changes the direction of the sort (handleSortDirectionChange).
handleSortChange is called anytime a new option in the dropdown menu is clicked, and it updates searchParam based on the option that was chosen.
handleSortDirectionChange works in tandem with the sortAscending state variables to keep track of whether the sort is currently displayed in ascending or descending order. There is a button that shows either an up or down arrow based on whether the sort is currently in ascending or descending order and clicking it will switch it and call the handleSortDirectionChange.

Phase 4

We implemented our visualizations using Recharts. In general, this involved calling the relevant API endpoints, collecting all of the data and sorting it based on what we wanted our visualization to display. For instance, one of our provider visualizations was the number of wins/goals/losses etc for each team, so we made the subsequent API call and scraped all of the information that we needed from it to store in a variable. We then used different Recharts components (based on the type of graph that we wanted to display) and the variable with the data to then display the information.

Challenges

Phase 1

One of the main challenges that we faced was learning frontend. Most of us didn't have that much experience with React, so there was a bit of a learning curve from trying to understand how React structures their design around the idea of components along with the same learning curve from just trying to understand HTML/CSS and knowing what properties to change and how to surround certain tags with containers in order to get the exact formatting desired. Most of this challenge was just overcomed by having to manually test out how assigning different properties or moving different components would generate changes in the website, so it was nice that React updated every time there was a code change since it made it very easy to just check what changes were made.
Another challenge that we faced was understanding how to best structure the entire project in order to be able to properly link together the different components. For instance, we had to decide how to best organize the model and instance pages, identifying that there is redundancy in structure amongst these but that they also have specific information tailored for each different model/instance.

Phase 2

One of the main challenges that we faced was properly setting up AWS Elastic Beanstalk. When setting up Elastic Beanstalk, there were a lot of additional setup steps (nginx, uwsgi) that weren't clearly stated in the AWS documentation on deploying a Flask app, and because of these hidden steps, a lot of time was spent understanding why the deployment was failing.
Another challenge we faced was with scraping all the information from the APIs. There was missing information that we had to handle as well as additional information that we had to scrape from other sources. Likewise, because of this missing information, it impacted how we were going to go about linking the different pages together.

Phase 3

One of the main challenges that we faced initially was restructuring of our backend API and how we interact with our backend API in order to handle the different requirements with searching/sorting/filtering. First, we had to add metadata to all of our responses because the searches/sorts/filters would now result in an unknown number of results being returned whereas previously we assumed how many instances we would have to display for each model. This was an issue because our pagination logic requires knowing the number of instances/pages in total, so we had to provide metadata in the backend that we could then pass into the pagination logic. Likewise, we also changed the way we were calling our backend API. Previously, we just had a single endpoint that we had to connect to, but with searching/sorting/filtering, we now ran into the issue of having multiple queries passed in and the best way to pass in those queries to the API backend call. Eventually, we decided to mirror the frontend URL to be what we needed to call the backend API with via searchParams.
Another challenge that we encountered was with writing tests. Getting the Selenium tests to run in the pipeline was a confusing process and because we ran out of CI/CD minutes, we were unable to run them towards the end because the Gitlab uses Linux machines which had all the services and dependencies that Selenium needed that a locally ran Gitlab runner did not have, so we were unable to run our tests in the pipeline. Additionally, there was a steep learning curve understanding how to write Jest tests.

Phase 4

We didn't really face many challenges in this phase since most of it was fairly simple. The only slight complication we encountered was with our provider's API not providing a parameter to control how many instances were returned in a page. This complicated the logic from making a single API call to having to collect a list of promises, so that we could wait for all of the API calls to return and update the variable a single time. Eventually, one of our group members also submitted a merge request to provide the capabilities to control the number of instances in a page, but since the idea of waiting on several requests/promises was a new concept, it was left in.