Technical Report

College Fit For Me

https://collegefitfor.me

Parth Shah
Leon Cai
Michael Munoz
Alec Hillyard
Chinedu Enwere
Jed Eloja

Table of Contents

Motivation

CollegeFitFor.me is a website that allows high school students to identify Universities that would be a good fit for them to attend based on a variety of needs and interests. Students are able to browse profiles of Universities containing general information such as tuition, acceptance rate, and post-graduation salary. Additionally, students can also view profiles of the cities that these universities are located in to better understand and factor in the dynamics of the area into their college decision. Popular attractions near universities are also listed to help students identify Universities that cater to interests such as good access to nature or frequent concerts.

CollegeFitFor.me encourages students to look beyond the surface level information of universities and craft an immersive college experience that facilitates the pursuit of their passions and goals outside of the classroom. Additionally, CollegeFitFor.me encourages students to plan for life after graduation and career prospects in the cities that these universities are located in and provides insight into the opportunities that these universities provide to get involved in these communities. Thus, for example, a Computer Science student considering UT Austin would learn not only that the school has a good Computer Science program, but also that Austin has a growing tech scene and that the university provides many opportunities to be involved in that scene through CS campus organizations, regular visits from industry leaders, and career fairs to connect students with internships.

By helping students build a college experience that is best for them, it is our hope that these students will be motivated to go above and beyond their responsibilities of schoolwork, taking leadership, bringing change that will positively benefit their student bodies, and making a positive impact in the communities that they live in.

User Stories

Our customers, Freshman to Freshgrad, submitted the following User Stories:

Create Restaurant Attributes

- Date Received: 2/25/20
- Specifications: Customer specified 10 attributes as a target amount
- Implementation:
  We created 10 attributes which included Name, Cuisine Type, Location, Operating Hours, Phone Number, Average Rating, Price, Accepted Currency, Online Order, Table Booking. We displayed this information on cards with the picture of the restaurant. These values are currently hardcoded in the React Frontend Code.
- Date Completed: 2/25/20
- Estimated Time: 2 Hours
- Actual Time: 1 Hour

Create University Attributes

- Date Received: 2/25/20
- Specifications: Customer specified 10 attributes as a target amount
- Implementation:
  We created 13 attributes which included Name, Acceptance Rate, Most Popular Major, Ownership, Tuition, Urbanization, Athletics Division, Average Alumni Salary, Graduation Rate, Retention Rate, Student-Faculty Ratio, Religious Affiliation, Demographics. We displayed this information on cards with the picture of the University. These values are currently hardcoded in the React Frontend Code.
- Date Completed: 2/25/20
- Estimated Time: 2 Hours
- Actual Time: 1 Hour

Create City Attributes

- Date Received: 2/25/20
- Specifications: Customer specified 10 attributes as a target amount
- Implementation:
  We created 10 attributes which included Median Age, Number of Sunny Days, Median House Price, Crime Rate, Economic Growth, Average Education Level, Political Leaning, Diversity, Air Quality Index. We displayed this information on cards with the picture of the City. These values are currently hardcoded in the React Frontend Code.
- Date Completed: 2/25/20
- Estimated Time: 2 Hours
- Actual Time: 1 Hour

Create Restaurant Instances

- Date Received: 2/25/20
- Specifications: Customer specified 3 instances as a target amount
- Implementation:

3 instances [McDonald’s, Wendy’s, Taco Bell] were introduced. They are hardcoded in the React Frontend Code. They are organized in card grid manner.

- Date Completed: 2/25/20
- Estimated Time: 2 Hours
- Actual Time: 1 Hour

Create University Instances

- Date Received: 2/25/20
- Specifications: Customer specified 3 instances as a target amount
- Implementation:

  3 instances [University of Texas at Austin, University of California-Berkeley, University of Washington-Seattle] were introduced. . They are hardcoded in the React Frontend Code. They are organized in card grid manner.
- Date Completed: 2/25/20
- Estimated Time: 2 Hours
- Actual Time: 3 Hour

Create City Instances

- Date Received: 2/25/20
- Specifications: Customer specified 3 instances as a target amount
- Implementation:

  3 instances [Austin, San Francisco, Seattle] were introduced. They are hardcoded in the React Frontend Code. They are organized in card grid manner.
- Date Completed: 2/25/20
- Estimated Time: 2 Hours
- Actual Time: 2 Hour

Rename City Attribute

- Date Received: 3/26/20
- Specifications: Label restaurants nearby cities as “Restaurants” unless non-restaurant city are added
- Implementation:

  We renamed the list of “attractions” near restaurants to “universities” because it was intended to be that way from the beginning. All “attractions” listed prior were universities. We renamed the list of “restaurants” near cities to “attractions” because some of the cities already had non-restaurant attractions listed nearby. These changes were made in the React Frontend Code.
- Date Completed: 3/27/20
- Estimated Time: 1 Hour
- Actual Time: 1 Hour

Rename Restaurant Attribute

- Date Received: 3/26/20
- Specifications: Label universities nearby restaurants as “universities” instead of as “attractions”
- Implementation:

  We renamed the list of “attractions” near restaurants to “universities” because it was intended to be that way from the beginning. All “attractions” listed prior were universities. These changes were made in the React Frontend Code.

- Date Completed: 3/27/20
- Estimated Time: 1 Hour
- Actual Time: 1 Hour

Create More Restaurant Instances

- Date Received: 3/26/20
- Specifications: Customer desired to have more instances of restaurants
- Implementation:
  2000 instances of restaurants exist after using the Yelp API to pull data on nearby restaurants to a predetermined list of 100 cities.
- Date Completed: 3/27/20
- Estimated Time: 4 Hours
- Actual Time: 4 Hours

Create More University Instances

- Date Received: 3/26/20
- Specifications: Customer desired to have more instances of universities
- Implementation:
  698 instances of universities exist after using the College Score Card API to pull data on nearby universities to a predetermined list of 100 cities.
- Date Completed: 3/27/20
- Estimated Time: 4 Hours
- Actual Time: 4 Hours

Create More City Instances

- Date Received: 3/26/20
- Specifications: Customer desired to have more instances of restaurants
- Implementation:
  2000 instances of restaurants exist after using Yelp to pull data on nearby restaurants to a predetermined list of 100 cities. 100 instances of cities were created by pulling data from the Numbeo API and Teleport API on a predetermined list of 100 cities. The list of 100 cities was arrived at after first using the 100 most populated cities in the United States, then changing out cities for alternatives if satisfactory data in the Numbeo and Teleport APIs was not present.
- Date Completed: 3/27/20
- Estimated Time: 4 Hours
- Actual Time: 4 Hours

Pagination Edge Cases

- Date Received: 4/13/20
- Specifications: City pages 6 and 7 cause the website to hang. Using the arrow to navigation pagination fails when loading the last page
- Implementation:
  We swapped from displaying 15 instances per page to 20 instances per page. The behavior of arrow buttons was also changed to refresh the last set of pages if trying to advance past the final set (or the first set of pages if trying to head back from the first set of pages).

- Date Completed: 4/14/20
- Estimated Time: 2 Hours
- Actual Time: 2 Hours

Confusion about "Number" Column for Universities

- Date Received: 4/13/20
- Specifications: The number displayed alongside nearby universities is too ambiguous in meaning.
- Implementation:
  We removed the numbers column in the display of nearby universities
- Date Completed: 4/14/20
- Estimated Time: 30 minutes
- Actual Time: 5 minutes

Confusion about the "Number" Column for Attractions

- Date Received: 4/13/20
- Specifications: The number displayed alongside nearby attractions is too ambiguous in meaning.
- Implementation:
  We removed the numbers column in the display of nearby attractions
- Date Completed: 4/14/20
- Estimated Time: 5 minutes
- Actual Time: 5 minutes

Implement Searching

- Date Received: 4/13/20
- Specifications: Provide a way to search the website for specific information
- Implementation:
  A search bar was implemented in each model page and an additional search bar was implemented at the top of the navigation bar. Using these search bars returns results grouped by model with matching text highlighted
- Date Completed: 4/14/20
- Estimated Time: 1 day and 7 hours
- Actual Time: 2 days

Edit landing page statements

- Date Received: 4/13/20
- Specifications: The old statement "Learn more different Universities, their surrounding areas, and attractions near them" needs to be replaced
- Implementation:
  The new statement displayed is “Learn about life at different Universities, their surrounding areas, and explore attractions near them.”
- Date Completed: 4/14/20
- Estimated Time: 15 minutes
- Actual Time: 5 minutes

We submitted the following User Stories to our developers, Pathogerm:

Add Link to GitLab Repository and API in About Page

- Date Submitted: 2/25/20
- Specification Text:

This is crucial as others trying to understand how to build a site like Pathogerm can refer to these resources. Ensure that the GitLab repo is public so anyone who clicks on the link can see the repository. Also, be sure to include all of the tools used for building the site in the about page as well as a quick description of each of them.

Add Description of Site in About Page

- Date Submitted: 2/25/20
- Specification Text:
  The about page should discuss the purpose of the site and what it seeks to accomplish. Be sure to discuss the intended type of users and how this site would benefit them. In addition to the description, also include a card for each developer describing a little about them and their role in the project.

Create 3 Instances of Years

- Date Submitted: 2/25/20
- Specification Text:
  When I click on the years' tab, there should be at least three different year cards that I can click on. For now, it is sufficient to hardcode the year cards, but in the future, they will need to be derived from an api. Each year card must have the name of the virus, a few details, and a link to the individual virus page.

Write a Technical Report using Grammarly

- Date Submitted: 2/25/20
- Specification Text:
  The technical report must describe the aspects outline in the project description including the motivation of the project, implementation of the API, and models used. Download the Grammarly extension so that grammar errors can be caught and corrected while writing the report. Keep in mind that the report should be written in the perspective that another group of Software Developers would know how to take over the project if necessary.

Design a RESTful API with Postman

- Date Submitted: 2/25/20
- Specification Text:
  At least 3 different datasets must be scraped when creating the API. Be sure to document the API on Postman well. This is crucial as the front end members of your team will need to easily understand how to use the API.

Increased clickability of cards

- Date Submitted: 3/26/20
- Specification Text:
  As a casual user, I want to be able to click on the cards to display more information so that my website experience is more intuitive. Currently, clicking on the cards instead of "More" causes the border to be highlighted. This causes me to think something should have happened, but nothing does unless I actually click on "More."

Card Length Variability

- Date Submitted: 3/26/20
- Specification Text:

As a person who wants this website to be popular, I want each card to be the same length so that a professional image is maintained. The cards already start on the same height for each row, but the varying lengths of each card is something that cannot be unnoticed once observed. Other users may feel the same and there are those whose website experience would suffer as a result.

Country Map Clickability Clarity

- Date Submitted: 3/26/20
- Specification Text:
  As a busy person, I want to have an easy way to differentiate between countries with additional data available (like the United States) and countries without (like Tajikistan) so that I spend less time trying to access information that I can't. I spent a lot of time trying to click on countries that did not have information available. In the current state, ignorant users like me could end up feeling frustrated if they keep on clicking something that they think can be clicked.

Southern Hemisphere Country Map Issues

- Date Submitted: 3/26/20
- Specification Text:
  As a frequent traveler, I want the map to be scrollable or scaled to fit my screen so that I can check out every country. Currently, I am unable to scroll down to click countries like Argentina, South Africa, or Australia. These countries are popular tourist destinations and assessing their ability to deal with diseases is important.

List Symptoms for Diseases

- Date Submitted: 3/26/20
- Specification Text:
  As a caring individual, I want to have some symptoms listed for each disease that is being tracked so I can look out for my own wellbeing. Given the current coronavirus outbreak, it would be helpful if people were more aware of what to look out for in case they suspect that they are sick with anything. Most of the time I am sick, it is not anything too serious, but I would like to know if I do potentially have anything more serious.

Increased Visual Clarity Map

- Date Submitted: 4/12/20
- Specification Text:
  As a person with bad eyesight, I want to have increased clarity with the map coloration. Currently, the orange highlight that happens when I hover a country blends in too well with the surrounding unselected red countries. I would appreciate having a different color scheme to have increased contrast between selected and unselected countries.

Cues for Displaying Country Information

- Date Submitted: 4/12/20
- Specification Text:
  As somebody who often fails to notice things, I want country information to be displayed more obviously when clicked on the map. Currently, I have to scroll down to look at country information after clicking. While it is nice that the information is being displayed, I did not notice the information initially and I

suspect many other users will fail to notice without some sort of cue to tell them to scroll down.

Increase Ease of Timeline Use

- Date Submitted: 4/12/20
- Specification Text:
  As a busy person, I want to have an easier way to choose the year in the timeline. Currently, I have to drag the indicator between 28 different years and finding a specific year can be cumbersome. I would appreciate having either a field to type in a specific year or having every available year number be shown at all times to speed up lookup.

Capitalize Attribute Fields in Tables

- Date Submitted: 4/12/20
- Specification Text:
  As a person who wants this website to be popular, I want each attribute to be capitalized in the disease table so that a professional image is maintained. Currently, only "Actions" is capitalized and the others aren't. Other users may feel the same and there are those whose website experience would suffer as a result.

Increase Visual Clarity Table Attributes

- Date Submitted: 4/12/20
- Specification Text:
  As a person with bad eyesight, I want to have increased clarity when observing tables like in the "Diseases" or "Country Data" tables. Currently, clicking on an attribute causes the text to become black and blend in with the background. I would appreciate having a different color scheme to have increased contrast between the selected attribute and website background.

RESTful API

Gitlab API

We used the Gitlab API to pull statistics from our Project Repository which are currently being displayed on the About Page. The statistics included issues and commits and are dynamically updated. We simply make a GET request to the API and the endpoint returns the necessary information in a JSON format. All logic for parsing this information is done in React.

RESTful API Documentation

We used Postman to design and publish our API. We have created 9 endpoints as a start to what we might need. We will add more endpoints in the future to serve the needs of the website.

Postman API: https://documenter.getpostman.com/view/10443615/SzKWtGrv?version=latest

GET - Cities
Example: https://api.collegefitfor.me/cities
This request returns all of the city instances along with all the attributes for each city

GET - City by Name
Example: https://api.collegefitfor.me/cities?name =austin_texas
Description: This request is a single city instance based on the name argument.

GET - City by Filter
Example: https://api.collegefitfor.me/cities/filter?state_name=Texas
Description: This request returns all of the city instances that have a specific attribute specified by the filter. In this case, the user asked to return all the cities that have more than 100 sunny days.

GET - City by Search
Example: api.collegefitfor.me/cities/search?search_query=texas
Description: This request returns the cities that have the search query in any field of the API object. In this case, this would return all cities with Texas in their name, all the cities with Texas as their state, etc.

GET - Universities
Example: https://api.collegefitfor.me/universities
Description: This request returns all of the University instances

GET - University by Name
Example: https://api.collegefitfor.me/universities?name=The University of Texas at Austin
Description: This request is a single University instance based on the university_name argument.

GET - University by Filter
Example: https://api.collegefitfor.me/universities/filter?city_name=Austin

Description: This request returns University instances based on the filter argument. In this case, this would return all the Universities in the city of Austin.

GET - University by Search
Example: api.collegefitfor.me/universities/search?search_query=texas
Description: This request returns the universities that have the search query in any field of the API object. In this case, this would return all universities with Texas in their name (i.e University of Texas at Austin), all the universities with Texas as their state, etc.

GET - Restaurants
Example: https://api.collegefitfor.me/restaurants
Description: This request returns all of the Restaurants instances

GET - Restaurants by Name
Example: https://api.collegefitfor.me/restaurants?name=Wendys
Description: This request returns a single Restaurant based on the name

GET - Restaurants by Filter
Example: https://api.collegefitfor.me/restaurants/filter?price=$$
Description: This request returns Restaurant instances based on the filter argument. In this case, this would return all restaurants with a price of "$$"

GET - Restaurants by Search
Example: api.collegefitfor.me/restaurants/search?search_query=texas
Description: This request returns the restaurants that have the search query in any field of the API object. In this case, this would return all restaurants with Texas in their name, all the restaurants with Texas as their state, etc.

Pagination

The pagination was built into both the backend and the frontend. For the backend, we paginated the query results from the database so each pagination call would only return 15 results. This made the frontend pagination simple and clean. By calling the REST API in this way "api.collegefitfor.me/cities?page=2", only 15 results were displayed per page. The frontend just iterates through the page argument in the API call. For the pagination buttons on the frontend, 7 page buttons are displayed at a time. Since there are only 7 pages for Cities, buttons for all 7 pages are simply displayed. However, for Universities and Attractions models which have more than 7 pages of items, the buttons for pages 1 through 7 are initially displayed. There is a forward arrow which when clicked will show buttons for pages 8 through 15 and automatically load page Clicking the backward arrow will show buttons for pages 1 through 7 again and load page 1, while clicking the forward arrow will show buttons for the next 7 pages with page 16 loaded. When the first 7 page buttons are shown, the backward arrow is not present. Conversely, when the last set of page buttons are shown, the forward arrow is not present.

Searching

Search functionality and highlighting was split between the backend and front end. The back end is solely responsible for returning models that match search query parameters. For each model, there is a separate search endpoint associated with the route name in the form "/:modelType/search" which takes in a "search_query" argument. The modelType field and search_query argument are specified in the front end based on the current URL. The backend API just checks if the search term is contained in any of the attributes of the model. If it is, it is added to the list of json objects that is returned.

The easiest way to implement highlighting was to then find the model parameter values that matched the search query. Routing was a bit of challenge at first since we had to update the window's location in one way or another upon each search query call. A front end search route consists of three parts in the form /search/:modelType?search_query=<search terms>". From this route, the router then mounts a new component called SearchResults. Upon mounting, the page then uses the model type and search query from the URL to make the api call. Upon a succesful request, models are then displayed mapped to Cards, styled and conditionally rendered based on model type, and arranged in a CardDeck. Search terms are highlighted in each Card with the help of a Highlighter component from the react package "react-highlight-words".

Filtering

Backend Filtering

Our backend filtering was fairly simple. First, we set up each model with a filter route (i.e /cities/filter). To boost performance, instead of sending in arguments to the various routes and changing the respective API request to return the new information, we just used SQL Alchemy's built in query.filter_by() function to go through the respective model database and return the result of the function as a json.dump(). To successfully parse the users filter query, we take in the user's options as a dictionary (**d) and unpack that dictionary into query.filter_by(). The results are in pages of 20 items each.

Frontend Filtering

For our frontend filtering, cities can be filtered by state and/or time zone, universities can be filtered by state, and attractions can be filtered by state and/or price level. By default, the filter values are set to "All" and filtering by a particular category does not take place unless a different value for that category is specified. One at least one of the filter values is changed, pressing go will display the new results. If one of the filters does not have "All" selected, the pagination buttons will change from showing five buttons at a time, to only showing the button of the first page, a forward button and a backwards button. This is because the number of pages of results available will depend on the search query. If there are no pages of results after the current page, the current page will simply be reloaded. If there are no pages of results before the current page, the current page will be reloaded. If all filter values are changed to "All" and the "Go" button is clicked, the original layout of pagination buttons will reappear.

Sorting

Our backend sorting was pretty simple due to SQL Alchemy's built sort_by function. This allowed us to combine the filter and sort so users could filter certain items and then apply some type of sorting on the results of the filter call. To add more sorting functionality to the API, we also allowed for both ascending and descending sorting (ex: /cities/filter?order=population*{asc} This allowed for frontend implementation flexibility. Currently, on the Frontend, the City and University attributes can only be sorted by ascending order, and the Attractions attributes can only be sorted in descending order. Because we used dictionary unpacking for the filter API which didn't allow for numerical based filtering, we made sure all the numerical data could be sorted.

On the frontend, the sorting can be applied in addition to specified filter options, or on its own with the default filter options of "All". If all of the filter options are set to "All", the layout of the pagination buttons at the bottom of the screen does not change. However, if one of more different filter options are selected in conjunction with sorting, the pagination buttons at the bottom change to show only the current page, a forward button, and a back button. The sort option can also be changed to "All", and selecting "All" then pressing the "Go" button will show the data unsorted with the current filter options applied if any.

Visualizations

Our Visualizations

Three visualizations were created to highlight trends in the data from each of our three models using the D3 visualization library. The first visualization was a Bubble Chart displaying how many attractions there were of each type. A new API endpoint on the backend was created to get all of the attractions categories, and the number of attractions in each category. The next visualization was a color-coded map of the US shading states by the number of universities they had. A new API endpoint was created to get the number of universities in each state, and on the Frontend, a list of each state's geobondaries was added to create the map. The last visualization was a bar graph displaying how many major cities each state had. A new API endpoint was created to get the number of major cities in each state.

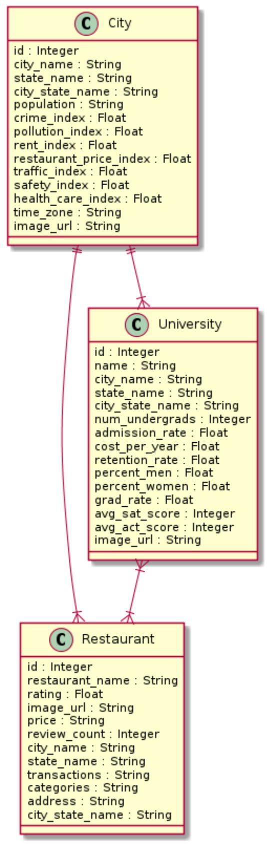
Customer Visualizations

Three visualizations were created for our customer site Pathogerm, highlighting trends in the countries that were used in their site. The first visualization was a Bubble Chart displaying how many countries were classified by a certain income level. Since the Pathogerm API was paginated and limited to showing 10 countries per page, the API was called in a loop to extract information on all countries. The number of countries in each income bracket were counted and then fed into the visualization. The next visualization was a bar graph showing the population of the most populous countries in their dataset. The same API was again called in a loop to extract information on all countries, and information on countries with a population of fifty million or greater were included. The last visualization was a pie chart showing the GDP share of all countries with a population of fifty million or greater. The same API was called in a loop, and the GDP values of countries with a population of fifty million or greater were fed into the visualization.

Database

We used the Postgres Database to store the 3 Tables, City, University, and Restaurant. Furthermore, we used SQLAlchemy to connect to the database and commit the Objects into the tables as rows. Furthermore, once the rows were committed to the tables, the backend developers used Postico, an application to view the data in the tables. This allowed us to make minor fixes in the data and make sure all the information was getting committed correctly. Initially, we used a PostgreSQL on the local server and then migrated all the information to an instance of Amazon Relational Database. We found it very easy to use SQL Alchemy's inbuilt ORM as it allowed us to easily define the scheme for each table.

The relationship between the tables can be seen in the Plant UML diagram. For each City instance, there are more than 1 University and Restaurant instances. For each University Instance, there is only 1 City Instance and more than 1 Restaurant Instances. For each Restaurants Instance, there is only one City Instance and more than 1 University Instances.



Testing

API Testing

For API testing, we used the Postman provided library to help us write our tests. In these tests, we checked that the requests had the appropriate status when queried, we made sure that a content-type header was present and that the content-type was JSON (to allow for easier parsing when queried), and that every successful request came back with at least one object of information. For our "by name" query test, we also created schema validation tests, to ensure that every "by name" query came back with the correct information and fields.

Backend Testing

For backend testing, we used the unit test class included in the Python Library. This allowed us to make sure that different components of the backend services were doing the proper things. In the tests, we query the tables in the database and make sure we have the right number of entries and correct schema. Furthermore, we also test helper functions that scraped information from API that seemed unreliable and not well tested. We plan to add more unit tests as we discover new ways to isolate the methods. Many of the functions commit new rows to the tables in the db so we did not test those.

GUI Testing

For GUI testing, we automated those tests with the Selenium WebDriver. These tests ensured that when we clicked a given button on a site, we were then redirected to the proper page. For example, to test that clicking the Cities button on the navigation bar does in fact redirect to the cities page, the GUI test clicked the Cities button, waited for the Cities page to load by sleeping for three seconds, and then confirmed that the page loaded was the Cities page by verifying the url of the page and text on the page. The WebDriver opens a new browser window with the site loaded, and clicks on various buttons on the site, later reporting the results of all of the corresponding tests.

Frontend Testing

For frontend testing, we used the Jest JavaScript testing framework. For each of the components of the site, we performed a Snapshot test to compare what that component should have looked like to what it actually looked like. Additionally, for many of the components, we tested to see if certain text elements appeared in the component as expected. For example, in the About page, after confirming that it rendered properly, we verified that there was a title component and that it was named "About Us" as expected.

Models

Our website features 3 models: Cities, Universities, and Restaurants. Each model is discussed in detail below. As we look into different data sources, the attributes for each model might be subject to change. All of the models are linked based on the city_name, state_name, and city_state_name attribute.

City
This model represents the cities that the universities in our site are located in.

- id : Integer
- city_name : String
- state_name : String
- city_state_name : String
- population : String
- crime_index : Float
- pollution_index : Float
- rent_index : Float
- restaurant_price_index : Float
- traffic_index : Float
- safety_index : Float
- health_care_index : Float
- time_zone : String
- image_url : String
- wiki_text: String

University
This model represents all of the universities that our site has profiles on for students to browse.

- id : Integer
- name : String
- city_name : String
- state_name : String
- city_state_name : String
- num_undergrads : Integer
- admission_rate : Float
- cost_per_year : Float
- retention_rate : Float
- percent_men : Float
- percent_women : Float
- grad_rate : Float
- avg_sat_score : Integer
- avg_act_score : Integer
- image_url : String
- wiki_test: String

Restaurant

This model represents the restaurants that are near the universities that our site lists.

- id : Integer
- restaurant_name : String
- rating : Float
- image_url : String
- price : String
- review_count : Integer
- city_name : String
- state_name : String
- transactions : String
- categories : String
- address : String
- city_state_name : String

Tools

React
We used React for our frontend and used BootStrap coupled with some vanilla CSS for the styling.

Cloudflare
For routing, we used Cloudflare to help us make our site secure (https). First, we changed the nameservers of the original website provided to us by Namecheap to the Cloudflare nameservers. After this was complete, we just added a CNAME record to allow our custom domain name to point to the one provided to us by our Elastic Beanstalk environment.

Docker
We used Docker to efficiently work across different machines. It also allowed us to easily deploy the website on AWS Elastic Beanstalk.

Amazon Web Services
We used Elastic Beanstalk, an Amazon Web Service, to host our website and the REST API. Beanstalk handled creating the EC2 instance as well as a load balancer.

Postman
We used Postman to design and document the RESTful API which we will be implementing in the next phase.

Gitlab
We used GitLab for our version control and issue tracker. The Gitlab API was used to keep track of statistics for each team member.

Slack
We used Slack to communicate with other members of the team. The Gitlab integration with Slack allowed us to stay updated with who was working on what feature.

Namecheap
We used this to obtain a free domain.

SQLAlchemy
We used this as an object-relational mapper and was an engine that allowed us to connect to the Database

Selenium
We used this to perform acceptance tests for GUI Tests

Postico
We used this to edit the table in the Postgres database

PostgreSQL
We used this as our relational database management system

Flask
We used this to create REST API and it served as a micro web framework

Mocha
We used this to test the JavaScript code which was the frontend

Hosting

For the first phase of this project, only the static website needed to be hosted in the cloud. We used Elastic Beanstalk, a service provided by AWS, to deploy our website. For the rest of the project, we also plan on using Elastic Beanstalk to host the frontend. Deploying the website using Elastic Beanstalk was made easy by using a docker image. In the Docker Image, we specified the tools we would need to be able to run the website on a “Virtual Machine”. For the purpose of the React frontend, we would just need to install Node.js which would allow npm to handle the dependencies specified in the package.json file located in the project directory. By letting Elastic Beanstalk CLI know that we are using Docker, it automatically sets up the environment to run the website. Elastic Beanstalk abstracts this process by setting up an EC2 instance, load balancer, and other services needed to host a website in the cloud. To host the backend, we used Elastic Beanstalk with a Dockerfile. In the Dockerfile we specified the dependencies such as Python and Flask related packages. Once the API was deployed on AWS, we rerouted the hosted URL or api.collegefitfor.me. Anytime we want to deploy new changes to either the REST API or Frontend we use the command “eb deploy” to deploy the new changes.