Technical Report

perfectfitfor.me

Motivation

Phase I: https://www.perfectfitfor.me/ is a website that allows for a mass congregation of data about different cities, including stats about that city, relevant transportation lines, and job listing in that city.

There are a lot of factors involved in determining where you want to live. For example, many parents would be concerned about the quality of education in a certain region, or commuters would be concerned about traffic ratings. Since there are so many data points involved in deciding where you want to live, we decided to make a website to display many important stats about different locations. Our website provides current job listings and links them to the cities they're in, as well as information about transportation in that city. Since many of us are planning on working at big companies after we graduate, we thought that having this website would allow us to have some insight into what place we would like to live.

Phase II: While continuing our focus on job listings and linking them with the cities they're in, to gain better insight into the place our customers would like to live and pursue a career, we have decided to absorb our transportation model into our city model and add the addition of an events model. We choose to do this because transportation and city go very hand in hand so, in the future, we will include transportation data on a per city basis. Now, this allows us to include another model on our website so our customers can better inform themselves about the places they want to live. The new addition of the events model, which we will further discuss in the next section, allows our customers to find a work-life balance that fits their needs in the cities they are considering.

Models

Phase I: We use three core models: jobs, cities, and transportation lines. We obtain the jobs from both Indeed's and Glassdoor's open source APIs. We obtain the cities from the Teleport API. Finally, we obtain the transportation data from the HERE API.

Relevant attributes about jobs include location, job title, type, job description, as well as several data points provided by the APIs.

Relevant attributes about cities include the cost of living, education, safety, travel connectivity as well a few other attributes that we may include.

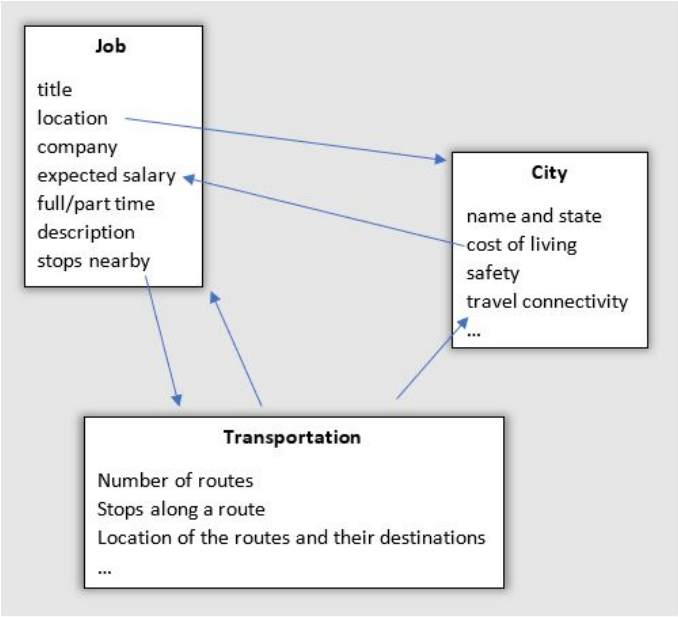


Figure 1

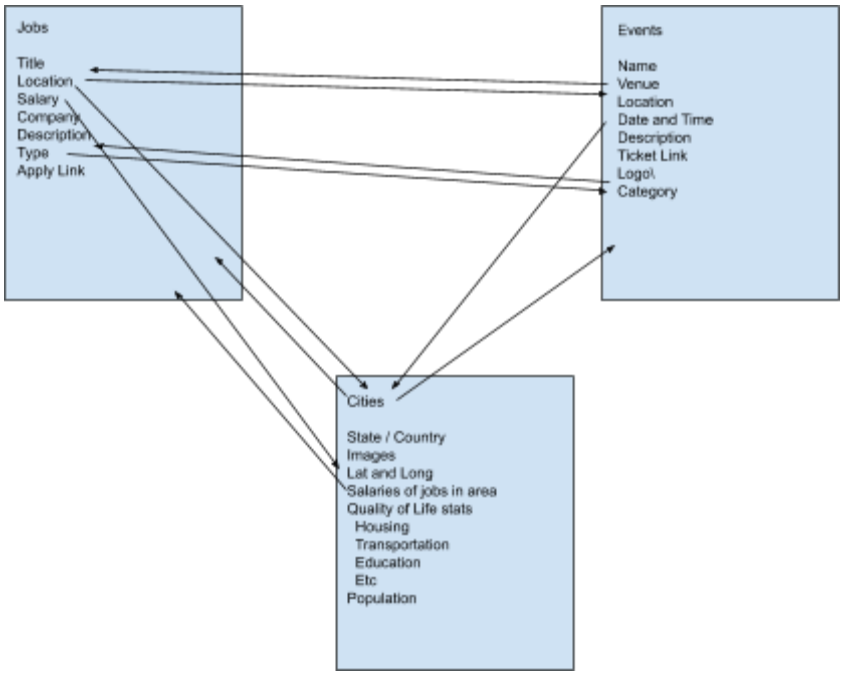
Relevant attributes about transportation lines include how many routes there are, the stops along the route and the location of the routes.

These models directly relate to one another. There is a relationship between jobs and cities naturally because jobs fall within cities. The attributes of cities show how good a city is, so a user can make a decision if a job is worth it based on the quality of a city. Transportation falls into this, because cities have various forms of transportation, and transportation may be an important factor in choosing a job or a city in which to live.

We are still debating whether we will keep transportation or not. In further phases, we may choose to switch to something else that more directly relates to cities and jobs. As of now, it fits.

Phase II: We have transitioned the transportation model to be absorbed by the city model. And we made the addition of an events model.

Relevant attributes about cities will now include commuting statistics about the city. The Teleport API and HERE API will be used to gather this information about the cities. Note, the Teleport API might be all that is necessary, and we may forgo the HERE API in future phases.



Relevant attributes about events will include the location, description, time, link to tickets, the hosting company, and putting the events into categories for easier user experience. This data will be collected using the Eventbrite API.

To relate events to jobs, we will show job listings in the area as well as jobs that we believe customers may be interested in because they are looking at a certain category of events. To relate events to cities, we will allow the customer to explore the city that the events are in so they can make an informed decision if the area is worth visiting or even settling down in. To connect cities to events, along with showing statistics about the city, we will show popular events in the area so our customers can get a peek about the culture of the city. And finally, to connect jobs to events, we will show events nearby the job listing so our customers can see if the area provides them a good work-life balance, as well as we will show events that may be in the interest of those looking at a certain category of jobs.

Phase III: Our UML diagram still remains relevant from Phase II into this Phase as no changes have been made to our models and in terms of interconnectedness.

Phase I: User Stories (As Developer and Customer)

As a consumer to Go Explore!:

- User story #1:
    - Make it so you don't have to click details to redirect to an instance. You could click anywhere in the box to redirect. It seems more intuitive
- User story #2:
    - Change the website title from 'React App' to something more meaningful
- User story #3:
    - I think you should try to put a container around the elements in your about page so that each person's descriptions have even dimensions.

- User story #4:
    - Maybe switch the instances' boxes so that the other attributes show instead of the description, so you can maintain the same sized boxes.
- User story #5:
    - On the tab in your browser, you should implement a custom browser icon instead of the default react one.

As a developer:

- User story #1: Dropdown button leads to an error page
    - This was an issue with a template we were using, and it has been resolved.
- User story #2: As a user, I would like to see more options on your home page.
    - As was mentioned in the closing comment, we plan on using a drop-down element to include visualization pages, but using drop-downs for the models does not make sense for us since there are many instances for each model.
- User story #3: The block in about page is not the same size
    - The problem was in the HTML formatting. We were able to completely fix this bug.
- User story #4: Mobile is not compatible yet
    - This was deemed as outside of the scope of Phase 1, but we will be implementing better mobile compatibility in the future. In the meantime, the user can ‘Request Desktop Site’ on most major mobile browsers.
- User story #5: The job model description
    - We understand the customers’ concern for more attributes on the job model, as we only met the 5 attribute criteria. As we get more familiar with the API and maybe look into other ones, we will address these additional attributes in the next phase.
- User story #6: The transportation page formatting
    - We fixed the spacing issues in the transportation page by adding appropriate margins and closed the issue.

Phase II: User Stories (As Developer and Customer)

As a consumer to Go Explore!:

- User Story #1: Make Page Bar Start on Page #1
    - Make sure Page Bar starts on page #1 when first clicking on model pages.
- User Story #2: Populate Instances’ Data on Model Pages
    - Remember to show attributes on each listing on the model pages.
- User Story #3: Change city’s instance page to show correct model names
    - Remember to change “modelName” to the correct names of your models, Activities and Facilities.
- User Story #4: Make Weather Attribute More Descriptive
    - Would like to see more than just an adjective describing the weather.

- User Story #5: More Attributes about the Facilities
    - Would Like to see maybe hours of operation, review, phone number, photos, etc.

As a developer:

- User Story #1: Description on Transportation Ratings
    - We will add a description of this statistic to our city's page that has absorbed this model. We have thus replaced the transportation model with events. We will address this in your other issue "Description on City's Stats" #2.
- User Story #2: Description on City's Stats (Estimate 1 hour, Incomplete in Phase II)
    - We're thinking about how to handle this by possibly adding text when you hover on that information that describes where we get those stats. Additionally, we might add some text on our listing or instance pages.
    - Because of some unexpected crunch time spent on fixing other issues, we will make this fix next phase. We understand this is a quick quality of life fix for our user experience and will complete this in the future.
- User Story #3: Multimedia (Estimate 6 hours, Incomplete in Phase II)
    - Note we completely agree with this especially for the jobs model. Sadly this will not be feasible this phase, so we will work towards this in the future.
- User Story #4: More Content in the Job Model (Estimate 5 hours, Incomplete in Phase II)
    - The front end should be able to add this easily. The problem is that our current APIs may not have enough content at the moment. We will have to look into this.
    - After some discussion, we weren't able to add enough more content that we were satisfied with this in phase II. If you could look at our jobs page after this phase, and update us on what you think. New Estimate: 5 hours
    - This issue will be readdressed in the next phase.
    - **Phase III:** We were able to close this user story. Look at User Story #6 in Phase III for more details.
- User Story #5: More Interconnectedness (Estimate 2 hours, Actual 3 hours)
    - Each event links to the hosting city and jobs nearby. Each city links to nearby events and job listings. Each job listing links to the city and nearby events.
    - Note we would like to connect jobs and events more closely by showing users instances that are in a similar category to their current viewing model instance.

Phase III: User Stories (As Developer and Customer)

As a consumer to Go Explore!:

- User Story #1: Need filter/search
    - There are no filter/search bars at the moment on any of the model pages
- User Story #2: About Page Broken
    - Get cannot GET when trying to access about page. About page was not loading when clicking on the button.
- User Story #3: Home Page Scrolling
    - Can scroll homepage to the left and right which is kind of weird. Would like to not have that side to side play when viewing website on fullscreen.
- User Story #4: Images on Instance Pages Stretched
    - Some of the images are stretched on the instance pages, mainly on the city page. Would like to see if more scaled and uniform.
- User Story #5: Cities have zip codes of 0
    - The cities have a zip code of 0. Would like to see it load the correct zip codes per city.

As a developer:

- User Story #1: Adding Filtering Function (Estimate Back-End: 2 hours, Actual: 4 hours) (Estimate Front-End: 3 hours, Actual 5 hours)
    - Here is how we implemented filtering. Jobs: Average Income, Degree Requirement, City Location. Cities: State, Cost of Living, Population. Events: City, State, Event Duration.
    - We are of course open to your suggestions on filtering attributes given our data.
- User Story #2: Adding Searching Function (Estimate Back-End: 3 hours, Actual: 4 hours) (Estimate Front-End: 3 hours, Actual: 5 hours)
    - We have implemented searching on each model page. Jobs has a google like search where the target string is checked against the job title, degree requirement, top cities, and description. Cities google like search was implemented where the target string is checked against the city name and state. Events google like search was implemented where the target string is checked against the event name, location, ticket link, and description.
- User Story #3: Adding Sorting Function (Estimate Back-End: 2 hours, Actual: 3 hours) (Estimate Front-End: 3 hours, Actual: 3 hours)
    - We have implemented sorting in the following manner. Jobs can sort on the job title in alphabetical and reverse alphabetical order. Cities can sort on the city names in alphabetical and reverse alphabetical order. Events can sort on the event names in alphabetical and reverse alphabetical order.
    - We like your suggestion but because our database at the moment does not have the overall ratings of the cities as that is done in the front-end. We, of course, look to add this in the future. And if you have any other sorting suggestions, please let us know!
- User Story #4: Clean Event Model's Data to Eliminate Repetition (Estimate Back-End: 3 hours, Estimate Front-End: 0 hours) (Incomplete in Phase III)

    - After discussion and understanding how events are pulled from the Eventbrite API, we at the moment decided to leave the event instances as is. The events appear as duplicates because the event is occurring in multiple locations. And to condense these duplicates into one event with multiple locations goes against how our model functions at the moment.
    - We will try to readdress this in Phase IV.
- User Story #5: Improve Pagination by Adding Previous Page Numbers (Estimate Back-End: 0 hours, Estimate Front-End: 3 hours) (Incomplete in Phase III)
    - Due to time constraints for completing the requirements of Phase III. We were not able to find time to add this functionality to our pagination.
    - We will look to readdress this in Phase IV.
- User Story #6: Adding at least Five Attribute to Job Model (Estimate Back-End: 1 hour, Actual: 1 hour) (Estimate Front-End: 1 hour, Actual: 1 hour)
    - We were able to pull more attributes from our Jobs API. Now it includes job title, description, type, average annual salary, education requirement, and top cities for the job in order of their salary earnings.

Phase IV: User Stories (As Developer and Customer)

As a consumer to Go Explore!:

- User Story #1: Errors when clicking on a Facilities tile
    - The link shows this error "Cannot GET /facility/1"
- User Story #2: Errors when clicking on an Activities tile
    - The webpage shows this error "Cannot GET /activity/1"
- User Story #3: Errors when clicking on a Cities tile
    - The webpage shows this error "Cannot GET /city/1"
- User Story #4: Add Data Visualization (1)
    - Make sure its a different type than Visualization (2) and (3)
- User Story #5: Add Data Visualization (2)
    - Make sure its a different type than Visualization (1) and (3)
- User Story #6: Add Data Visualization (3)
    - Make sure its a different type than Visualization (1) and (2)

As a developer:

- User Story #1: Differentiate Two Search Bars (Estimate: 1.5 hour, Actual: .5 hour)
    - After discussion, we decided the way we would like to differentiate the search bars, is to add a placeholder text in the search box that informs the user whether they will be searching over the entire site or one model.
- User Story #2: Arrange Sorting Filtering on each Page (Estimate: 1 hour, Actual: 1 hour)
    - We adjusted the filtering and sorting to look like a cleaner design. But we decided to keep these features across the top of the page.

- User Story #3: Multimedia on Job Page (Estimate Back-End: 3 hours Estimate Front-End: 2 hours) (Shelved)
    - Under the constraints of this final phase, we decided not to address this issue. We believe our jobs instance page provides adequate information as is. The only useful addition would be to include videos, but we were not able to find an adequate API for this.
    - Issue Shelved.
- User Story #4: Dynamic Links on Home Page (Estimate: 15 minutes, Actual: 5 minutes)
    - We added dynamic links to the pictures on the Home page. They now redirect to the corresponding model page.
- User Story #5: Create Sliding Window on Home Page (Estimate: 2 hours) (Shelved)
    - After discussion, we decided not to implement this. The novelty of adding a sliding window of images did not appeal to us. If we could maybe build upon this and make each window interactable or redirect to another part of the site, then we could see the merit. But this addition is outside our scope for this phase.
    - Issue Shelved.

RESTful API

API Documentation: https://documenter.getpostman.com/view/6807504/S17wNmPd

GET requests will return a list of instances based on which model is requested. With a unique ID, a particular instance of a model can be returned. All requests will be returned in JSON format.

Endpoints:

- api.perfectfitfor.me/cities
  returns list of cities
- api.perfectfitfor.me/cities/state/<state>
  returns list of cities in a certain state
- api.perfectfitfor.me/cities/page/<page>
  returns a single page of cities
- api.perfectfitfor.me/jobs
  returns list of jobs
- api.perfectfitfor.me/jobs/id/<id>
  returns a single job by id
- api.perfectfitfor.me/jobs/page/<page>
  returns a single page of jobs
- api.perfectfitfor.me/events
  returns a list of events
- api.perfectfitfor.me/events/page/<page>
  returns a single page of events

- api.perfectfitfor.me/<model>/filter/<attribute>/<value>
  returns model filtered by attribute with value
- api.perfectfitfor.me/<model>/search/<query>
  returns model searched with query
- api.perfectfitfor.me/<model>/search/<query>/<page>
  returns a single page of model searched with query
- api.perfectfitfor.me/api/<model>/sort/<attribute>
  returns model sorted by attribute
- api.perfectfitfor.me/api/<model>/sort/<attribute>/<page>
  returns a single page of model sorted by attribute
- api.perfectfitfor.me/<model>/desc_sort/<attribute>
  returns model sorted descendingly by attribute
- api.perfectfitfor.me/<model>/desc_sort/<attribute>/<page>
  returns a single page of model sorted by attribute

The APIs used when collecting data were the CareerOneStop API for information about jobs, Teleport API for information about cities, and the Eventbrite API for information about events.

Tools

Our front end currently uses React.JS for making more modular components for our website, and React Bootstrap and React Router for making writing the React code more convenient for us, as well as to make our website look better and render components quicker.

The back end uses Python, Flask, and SQLAlchemy as its framework. The API is documented with Postman. The MySQL database is stored using Amazon RDS.

Hosting

Amazon Web Services (or AWS) is a suite of various services in cloud computing. It was chosen due to its popularity in the field. Amazon S3 is a cloud storage service that also offers static web hosting. Amazon CloudFront is a content delivery network that improves our website's availability and performance and provides HTTPS services. Amazon Elastic Beanstalk is an application deployment service that simplifies the process of hosting a web app. All of these services were chosen due to their integration with AWS. Namecheap is a domain name registrar, chosen due to its cheap prices.

Our custom domain (https://www.perfectfitfor.me/) was obtained through Namecheap and is linked to our Amazon CloudFront domain (https://d20zyjv6hzsjz6.cloudfront.net/). This domain is then linked to our Amazon S3 domain (http://perfect-fit-for-me.s3-website.us-east-2.amazonaws.com/), which is where our website resides. The backend is hosted through Amazon Elastic Beanstalk (http://perfectfitforme-env.bdibh8r7gh.us-east-2.elasticbeanstalk.com/api/). A

subdomain of our custom domain (https://api.perfectfitfor.me/) is linked to the Elastic Beanstalk domain.

Pagination

- **Back-end:**
    - Inside the backend, we've created GET requests for each model that populates all of the data for that specific model, and then for each model, we have GET requests for getting info about a specific instance of that model. The GET request for cities returns a single dictionary, while events and jobs return an array of dictionaries.
- **Front-end:**
    - We first fetch all of the data for each model. After we have the data for each model, we split all of the listings into an array of listing pages that contain 9 elements each. We then used the React Bootstrap Pagination component to render the page bar and using the active page number on the pagebar (which is obtained by parsing the URL) we display the listing components at that pages index in the array.

Database

The MySQL database is hosted on Amazon RDS, a relational database service offered by Amazon. We chose Amazon RDS due to its integration with AWS. It contains three tables: cities, jobs, and events. SQLAlchemy is used to query and modify the database through Python. The information to access it is {username: "perfectfit", password: REDACTED, endpoint: "mysql-db-instance.chdg6as3bxgl.us-east-2.rds.amazonaws.com:3306", database name: "perfectfitdb"}.

The cities table stores instances of the city model with the attributes of id, images (mobile/web), location (latitude/longitude/state), population, and qualities (commute/cost of living/housing/tolerance). The jobs table stores instances of the job model with the attributes of job title, annual salary, id, description, and location (city/state). The events table stores instances of the event model with the attributes of eventid, name, summary, address, city, state, venue, start, end, timezone, URL, and logo.

Testing

- Location
    - There are multiple sets of tests in our GitLab repository
        - The front-end tests are located in ./frontend/src and ./frontend/src/views in files named *.test.js
        - The back-end unit tests are located in ./backend/tests.py while the Postman tests are located in Postman.json
- Contents
    - The contents of the tests are divided as follows:

        - The front-end tests primarily test if an instance of our views will render correctly, and only creates one instance. We did the tests using enzyme which was made specifically for testing react code
        - The back-end tests verify that the built-in methods of the model classes work as intended and return the right output. They also test queries of the database. The tests verify that search, filter and sort actually search properly, filter by the correct attribute and sort in the correct order. The Postman tests confirm that the API calls behave as expected.
- **Mocha**
    - Our front-end supports testing using Mocha, however, we got confirmation that we were allowed to use enzyme to test our react code instead, so we primarily used that
- **Enzyme**
    - Enzyme is the primary tool our front-end uses for sanity checks on the react code we wrote. If any of the tests fail, we can source the error back to the specific view, so it makes it easier to figure out what needs to be fixed.
- **Selenium**
    - Used to create acceptance tests for the GUI of our website. Selenium is used to call up a web browser which we use to call specific URLs, make sure buttons work, a text is displaying correctly, etc. With these tests, we can be sure our UI is loading our web pages properly.
- **Postman**
    - Postman is used to test our API. It ensures that requests are answered correctly and that the contents behave as expected.

Filtering

- **Back-End:**
    - We implemented an API call with the route /api/<model>/filter/<attr>/<value>. For this call, we parse the model, attribute, and value parameters for valid input, and then trivially query the database accordingly. For example, we can filter cities by population by simply checking that the population attribute is <200000.
    - For each model, there are three attributes you can filter by.
    - Jobs:
        - Income
            - Can have values 1, 2, 3, 4, 5
            - 1 represents a salary of less than 30000
            - 2 represents a salary between 30000 and 50000
            - 3 represents a salary between 50000 and 70000
            - 4 represents a salary between 70000 and 90000
            - 5 represents a salary greater or equal to 90000
        - Education (edu)

                - Can have values bac, mas, phd
                - Bac represents a Bachelor's degree
                - Mas represents a Master's degree
                - PhD represents a Doctoral or professional degree
            - Location (loc)
                - Queries through five top paying cities to filter by one of those five
                - Has a value of city which corresponds to the city name
    - Cities:
        - Cost of Living (col)
            - Can have values 1, 2, 3, 4, 5
            - 1 represents a cost of living score of 2 or less
            - 2 represents a cost of living score of (2, 4]
            - 3 represents a cost of living score of (4, 6]
            - 4 represents a cost of living score of (6, 8]
            - 5 represents a cost of living score of (8, 10]
            - Note: While cost of living is out of 10, there are no cities between 8 and 10
        - Population (pop)
            - Can have values 1, 2, 3
            - 1 represents a population of [0, 200000]
            - 2 represents a population of (200000,999999)
            - 3 represents a population of [1000000, ∞)
        - State
            - Has a value corresponding to the state's name
    - Events:
        - City
            - Has a value corresponding to the city in which the event is taking place
        - State
            - Has a value corresponding to the state in which the event is taking place
        - Duration
            - Can have values 1, 2, 3
            - 1 represents a duration of [0, 1)
            - 2 represents a duration of [1, 4)
            - 3 represents a duration of [4, ∞)
- **Front-End:**
    - Filtering required a little bit of extra logic in the front-end. Firstly, we needed to have different dropdowns for each attribute we wanted to filter by. This was not particularly difficult, but for each of the filters we needed to figure out where the site would route to, and based off of that, fetch different data from the backend. Filtering is split up by each model.
        - **Cities**

        - For cities, we allowed the user to filter by city, population, and cost of living. Filtering for each of these attributes brings the user to a different page that specifically fetches the data based of what filter the user chose.
    - **Events**
        - For events we allow filtering by city, state, and event duration. Duration is defined as the length of the event in hours.
    - **Jobs**
        - For jobs we allow filtering by city, income, and degree requirements. The income for a job is filtered by average income, but the job listings vary across multiple states and cities. The city filter attribute looks through all of the cities that a job listing is present in and will only display the job listing if the city the user typed is in that list of cities.

Searching

- **Back-End:**
    - We implemented an API call with the route /api/<model>/search/<query>. This call parses the model, then searches every relevant attribute for every row of that model in the database for the query string using the LIKE operator and % wildcards.
    - When model is passed as ‘all’, we conduct a search of all models by using the same logic for individual models, and adding each resulting list to a dictionary with the model name as the key.
    - Model must be one of the following {cities, jobs, events, all}. Query can be any string.
- **Front-End:**
    - For searching on the front-end we created an entirely different view for each of the models to load different listings. The listings on the searching pages highlight relevant search terms by checking the attributes of the model to see which attributes contained the string. None of our searching implements paging since it was not in the specifications whether we needed to or not, and we feel our site feels better without paginating search results. In particular, there were two types of search views
        - The first of such was the search for the entire site. This view displays all the search results for cities at the top, then below those, displays all the search results for events, and then below that, displays the search results for jobs. Each of the cards highlight all places where the searched term is present making it easier for the user to spot where exactly their search appeared.

        - The second type of view was the searches for the individual models, which behaved in the same way as the search for the entire site, but they only displayed instances of that particular model.

Sorting

- **Back-End:**
    - We implemented two APIs call with the routes /api/<model>/sort/<attribute> and /api/<model>/desc_sort/<attribute>. These API calls trivially use the ORDER BY keyword after confirming the model and attributes are valid entries. The descending sort simply tacks on the DESC keyword.
    - Model must be one of the following {cities, jobs, events}. Attribute requirements are as follows.
        - For cities: {id, name, state, population, latitude, longitude, housing, cost_of_living, tolerance, commute}
        - For jobs: {job_id, job_title, description, education, salary, city1, city2, city3, city4, city5, salary1, salary2, salary3, salary4, salary5}
        - For events: {eventid, name, summary, address, city, satte, venue, start, end, timezone}
- **Front-End:**
    - Sorting in the front-end was a pretty trivial task for us. We implemented sorting by name, so the logic for doing pagination was almost identical to the logic for pagination on the normal model listing pages. The main thing that needed to be added was a dropdown menu for the user to select whether they wanted to sort in ascending order or descending order. After that was determined, we just had to do a little bit of URL parsing to determine where we needed to fetch our data from so the correct instances would load depending on what option the user sorted by.

Visualizations

Our visualizations can be found by clicking the links in our navigation bar under the "Visualizations" dropdown.

United States of America Map: Hot Zones of Events

- Our map visualization shows the number of events planned and currently happening at a given state. You can see the total number of events by hovering over the state or visually the more events at that state will color more red correspondingly, representing a “hotter” state. With this visualization, we show the popular states with events that our site can provide information to the user. And with this information, they can filter by state to their desires, when visiting our events model.

Pie Chart: Breakdown of Job Education Requirements

- Our pie chart visualization is meant to show a breakdown of how many jobs require different types of degrees. The reason why we chose this visualization is because we wanted to illustrate the education requirements for many of the jobs our site includes. This is important because when determining where you want to live, you need to understand that even if there are a lot of jobs available, you need to have the proper education to work at the jobs present at those locations. We want our users to have a general understanding of what jobs our site provides exposure to, and the options they have based off of what level of education they have.

Bar Graph: Histogram of Overall City Ratings

- Our histogram is meant to show the distribution of overall ratings of cities that our site contains. The main reason for choosing this visualization is so that people can get an idea of where each of the particular cities they're looking at compare to other cities. We think that this is important because it's important to evaluate scores relative to other scores, so we hope the users will have a better understanding of what it means to have a certain rating if they have an idea of what the distribution looks like.

Self-Criticism

Phase I:

- **Back-End:**
  - What did we do well?
    - We came up with a solid idea for our website. The original concept was to act as a job search website that provides more information about the city and other aspects outside of the actual job that are important.
  - What did we learn?
    - We learned how to access APIs and the general format of their calls, documentation, and responses. We also got a glimpse of how the services provided by AWS operate and what we would have to use in the future.
  - What can we do better?
    - We could have looked into the availability of APIs better and realized that publicly available APIs for job listings are not a thing. On the contrary, many job websites pay to acquire these listings.
  - What puzzles us?
    - For the back-end, there weren't too many complications due to the tutorial for setting up a static website for a React app

hosted on S3. Postman seemed overwhelming at the start, but simplified considerably when we started to understand the API-specific terminology.

- **Front-End:**
    - What did we do well?
        - For phase one, I thought that we had a pretty good design for what our listing and instance pages would look like. I think that the design we decided on for our site ended up looking pretty good, and we have not really had to update the overall aesthetic of our site due to the effort we put into this phase
    - What did we learn?
        - Since this phase was a lot of front-end, we learned a lot of things. We decided to start with React on this phase because it would save time later on, and so we learned a lot of things about how the React lifecycle works. This phase contained a lot design in regards to how we wanted our website's interface to look, so we also got some exposure to making a website design.
    - What can we do better?
        - To be completely honest, because we went into this project with not too much experience, some of the CSS and React code has been pretty messy. We could have definitely done a better job of making more modular React components, and organizing our code, and our files.
    - What puzzles us?
        - The biggest challenge for this phase was getting used to React, and just creating a design that we thought looked pretty good. There was a lot of thought put into how we would display each of the instance pages and listing pages in an engaging way that looked professional, and we had a bit of trouble deciding how to tackle all of that within the time constraints.

Phase II:

- **Back-End:**
    - What did we do well?
        - Due to the extensive effort put into setting up our application and database, it made it much easier to update them in the future. Deploying an updated version of the Flask application was as simple as creating a zip and uploading it to AWS. Updating and querying the database was as simple as a line of code in our application.
    - What did we learn?

        - This phase was much more involved when it came to the AWS services and the database. We had to learn how to create an application using Flask and deploy it through Elastic Beanstalk. We also had to set up the database by creating tables and populating them with instances. Furthermore, we had to then connect the two by handling API requests to our application by querying the database.
        - Trying to change the attributes of a table without deleting the table first causes problems, and you have to refresh the database to fix it.
    - What can we do better?
        - We should have put more thought in determining what attributes our models would require. We ended up having to drop the tables and recreate them after scraping the APIs another time because we had no way of easily conveying the information without another attribute.
    - What puzzles us?
        - It's worth it to be pedantic when dealing with databases. A lot of issues arose when we had the mishap of changing parts of our model classes before deleting the entries in the table or dropping the table altogether.
- **Front-End:**
    - What did we do well?
        - For this phase we really did not have all that much to do with our design but we added pagination. We think that the listing pages ended up turning out pretty well, and that most of our models look pretty good.
    - What did we learn?
        - This phase taught us a lot about how to fetch data from our backend API. We did not know how much of a pain it could be to load in data dynamically from a JSON format, but we do feel like we have a better understanding of the React lifecycle after this phase.
    - What can we do better?
        - We think we could have organized our pagebar better. Most of the stuff we worked on this phase ended up looking fine, and our pagebar is no exception, but we had to do some weird stuff in our code to make it work as we intended. We wish we could have spent more time fixing it and making the code cleaner, but ultimately time constraints did not allow it.
    - What puzzles us?
        - What really gave us trouble this phase was getting our backend running on HTTPS. This stressed us out because on the localhost our code was working, but when we deployed our site we realized the the SSL certificate for our

backend was invalid, so our website was not actually able to fetch the data from our API. Fixing this took a lot of time, but we now understand the importance of making only HTTPS requests from an HTTPS site.

Phase III:

- **Back-End:**
    - What did we do well?
        - Through trial and error, we came up with a clean separation of our files that made deploying much faster and made accomplishing specific tasks much easier. For example, scraping the APIs again to update our table was done by simply compiling and executing one file.
    - What did we learn?
        - We learned how to query our database effectively to acquire the information we actually wanted. Implementing filter, search, and sort boiled down to how we could selectively gather the relevant instances and compile them into a straightforward structure.
    - What can we do better?
        - We should have decided on a standard format for our JSON returns and for our code in general. We ended up having to put in more work down the line when things required changing or fixing that could have been avoided.
        - It also may be worth it to the query parameters, so you can filter by more than one thing. With the current implementation, it becomes overbearing if you want to try and have multiple queries since you would have to reroute for every combination of different filters
    - What puzzles us?
        - We had a significant hiccup when we realized our test environments did not match what someone on production might be using. Two problems that occurred from this was that our API was not properly handling HTTPS requests and that some of the API calls worked properly when tested locally but not when deployed on AWS.
- **Front-End:**
    - What did we do well?
        - For this phase we think our searching and filtering came out to be better than we expected. We got a bit of a late start on this phase, so we had to rush a lot of things on the front-end, but despite that we feel like our highlighting and search results ended up being pretty decent.
    - What did we learn?

        - We learned a lot about how searching and highlighting works, and how to use HTML input tags to store information that the user can type in.
    - What can we do better?
        - We felt like our search results page was adequate, but not quite what we wanted it to be. There was some problems with how the highlighting looked on different browsers, and we did not really get to fix that.
    - What puzzles us?
        - What really challenged us this phase was just deciding how we would structure our search and filter results. It was hard deciding how exactly we would differentiate the search results pages from the listing pages.

Phase IV:

- **Back-End:**
    - What did we do well?
        - There was not a lot of refactoring to be done because we caught on to our mistake of not deciding the structure of our code beforehand. Thus what we did well was taking the preventive measure of designing our code to avoid excessive clutter in the future.
    - What did we learn?
        - This was something we gradually improved throughout the phases, but we learned the importance of communication and documentation. Slack and Gitlab issues made it much easier to streamline our efforts on tasks that had more priority.
    - What can we do better?
        - We could have used our time more efficiently by delegating tasks beforehand. For this phase, we ended up dealing with all of the issues relatively close to the deadline which made things more stressful overall.
    - What puzzles us?
        - Although in general I believe this team performed greatly, it's surprising how quickly things can become hectic. It makes us more appreciative of the effort that's been put into delegating roles in the industry.
- **Front-End:**
    - What did we do well?
        - We think we did a pretty good job of making our visualizations relevant to the core idea of our site. Also we did some cleaning up to make the code better, which is always nice.

    - What did we learn?
        - We feel like we learned a lot about how D3 works, and how to effectively collect and visualize data from a different API.
    - What can we do better?
        - We think we could have done a bit better at cleaning up the code, but there was so much to refactor that we just tackled the big things. We feel like under ideal circumstances we would have been able to make our site more optimized as well, but we just did not have the time or resources.
    - What puzzles us?
        - The thing that gave us a bit of trouble this phase was understanding how to get the D3 visualizations to look good on our site, as well as adding the dropdown to our navigation bar that allowed the users to choose which visualization they wanted to see.

Criticism for Our Developers - Go Explore! (xplorator.me)

- What did they do well?
    - Their website has definitely come along way. It has a nice splash page. Their drop down menus for sort, filter and search on the list of models pages are a nice addition.
    - They listened to our user stories pretty well.
    - I like that they say how many activities are happening in a cit
- What did we learn from their website?
    - The importance of working code and testing code
    - How facilities and activities link together and tie into what activities you can do in a city
- What can they do better?
    - Have working instance pages
    - A nice quality of life improvement would be to redirect “https://api.xplorator.me” to the Postman documentation or some other API documentation.
    - Their city API call that’s supposed to return a list of all their cities only returns one page worth of cities.
- What puzzles us about their website?
    - None of their instance pages actually work
    - When you are routed to a list of models page and click an instance, you are redirected to a page that says “Cannot GET /<insert route here>.” If you try and hit back on the browser it just gives you that message for all of the routes, even ones that were working previously.