Group #22
Larry Ukaoma
Lloyd Gumireddy
Edmond Gagnon
Ian McDonald
Douglas Archibald

Motivation

The goal behind Costly Commute is to help people find the best home possible given a workplace from a single website. Of course, not everybody is looking for the same things in a home. Values change from person to person - some people may prefer convenience over luxury, and everyone has their own budget to work around. With our website, we hope to provide all the information they need through a user-friendly experience that makes the process of finding a new home as painless as possible. Our main demographic is graduating or recently graduated college students evaluating their housing options, so we're going to be showing only renting options because we believe students aren't going to want to buy as soon as they leave college. We also wanted to promote the use of public transport in the commute because bussing is cheap, reduces traffic on the road, and leaves a smaller carbon footprint.

Developer stories

Phase 1 Stories

1. Get a domain. (Estimated 30 minutes to complete.)
2. Postman API Schema - Detailed under the RESTful API. (Estimated 3 hours to complete.)
3. Set up AWS to host - The site is deployed on AWS Amplify static hosting. To update the public site, merge into `amplify-console`. (Estimated 1.5 hours to complete.)
4. Create the static pages of the website - Most, if not all, of the pages, have been organized using Bootstrap elements for better and more uniform viewing across browsers of all sizes/types. Each of the models is accessible from the navigation header, listed under the links 'Commute,' 'Crime,' 'Housing,' and 'City'. Each link will take you to the respective model's grid, where the properties of each instance are listed on a single page. Since we don't have a proper database to request from yet, the data is initialized inside each page and kept as a constant. Each "Read More" button will take you to that instance's own page by storing identifiers in the URL, using the passed identifiers to query it again from the constant and displaying it on the page. This framework will be useful later in the development process and easier to refactor when we have a database to query data from. (Estimated 3 hours to complete.)
5. Create the About page - To create an About page with dynamic updates to each member's commits and issues, we fetch the data using a public/private SSH key to

access the repository through the GitLab API, and map the objects we receive from the call to each person by the e-mail associated with their GitLab account. This data feeds into the HTML elements on the About page for the user to see. (Estimated 2 hours to complete)

Phase 2 Stories

6. Add more instances per model - We scraped from our APIs for a set list of cities that we wanted our website to feature and stored them in our production database. (Estimated 10 hours to complete)
7. Add pagination - See Pagination. The back end returns results broken up into pages for a query, and the front end can further break up the received information into batches that are displayed across tabs on the model pages. (Estimated 5 hours to complete)
8. Add media to instance pages - For the housing, the page presents an image and a map location pinpointing the house, the transit page has an image of the specified route and a picture of the endpoint, and the city has several images retrieved from Wikipedia and an overall map of the city's coordinate center. (Estimated 15 hours to complete)
9. Format data on instance pages - This is rather subjective, but the model information is listed in a much more presentable way. (Estimated 5 hours to complete)
10. Link models to other models - This is done in the backend through the one-to-many and many-to-many relationships supported by SQL's foreign key and association table features. Each housing instance is tied to a city, and can have many different routes. Each city instance has a list of available housing, and a list of transport routes for that city. Each transit instance is tied to a city, and has a list of housing that often uses that route. The front-end uses this information to connect information and link to related models. (Estimated 10 hours to complete)

Phase 3 Stories

11. Add a description of your website to the About page - Since all we had to do was add a description of the website to the About page, this wasn't too hard to handle. (Estimated 15 minutes to complete)
12. Page numbers are incorrect on the models - The "next" button skipping 4 pages was a semantic issue, and wasn't too hard to fix either. (Estimated 15 minutes to complete)
13. Make search form specific to each model - See Searching/Filtering. For searching to work properly, we had to refactor the back-end API to fit the Flask RESTless model. Once the back-end supported searching/filtering, the front-end uses it to find all result matches for each specific model. (Estimated 20 hours to complete)
14. Remove instance information when selecting instance from list page - The search form is now hidden until prompted by a button click, so the user shouldn't be overloaded with information on the model listing pages. (Estimated 5 hours to complete)
15. Fix filtering/searching and allow sorting by attributes - See Searching/Filtering or issue #12. Filtering/searching/sorting came with switching the back-end to the RESTless

model, and is done by user selection in the query panel on the front-end itself. (Estimated 20 hours to complete)

Customer stories

Our developers are Resonance, who are developing a music-based social media site which can be found here.

Phase 1 Stories

1. Be able to access all the pages of a website - as a customer, we would like all the pages that the developers have worked on to be tangible or visible. (Estimate: < 1 hour)
2. Link the albums listed to their respective instances - being able to read about albums in more detail is important. (Estimate: < 3 hours)
3. Pictures on instance pages - Illustration goes a long way in making a website easy to look at. (Estimate: < 1 hour)
4. Card for concert prices - showing the price of a concert in an easy-to-view place is very important to a customer. (Estimate: < 2 hours)
5. See a list of friends on different users - showing the connections between users is important for an app focused on linking people with similar tastes. (Estimate: < 3 hours)

Phase 2 Stories

6. Create unit tests and acceptance tests for the website - we want to be sure that the website does not fail easily and can handle user errors. (Estimate: < 2 hours)
7. Update the website from static information to dynamic information - the website needs to be able to support pulling data from a database and displaying different models and instances. (Estimate: < 6 hours)
8. Containerize the application's front-end and back-end using Docker - isolation would be better for scalability. (Estimate: < 3 hours)
9. Create a single instance page per model to display variable information from different instances - we don't want to create files that rely on individual instance information because that would not scale well with space. (Estimate: < 10 hours)
10. Use the APIs to retrieve information that may not already be present in the database - given the scale of the project, it may not be wise to store all the information from the APIs in one go. It would be better to use the database as a cache of sorts, and whenever a miss occurs, the application should try to retrieve the requested information directly from its APIs. (Estimate: < 14 hours)

Phase 2 Assessment

The team did fairly well with setting up their environments, but the actual website behavior was unobservable because the pages would throw errors whenever they were accessed.

Phase 3 Stories

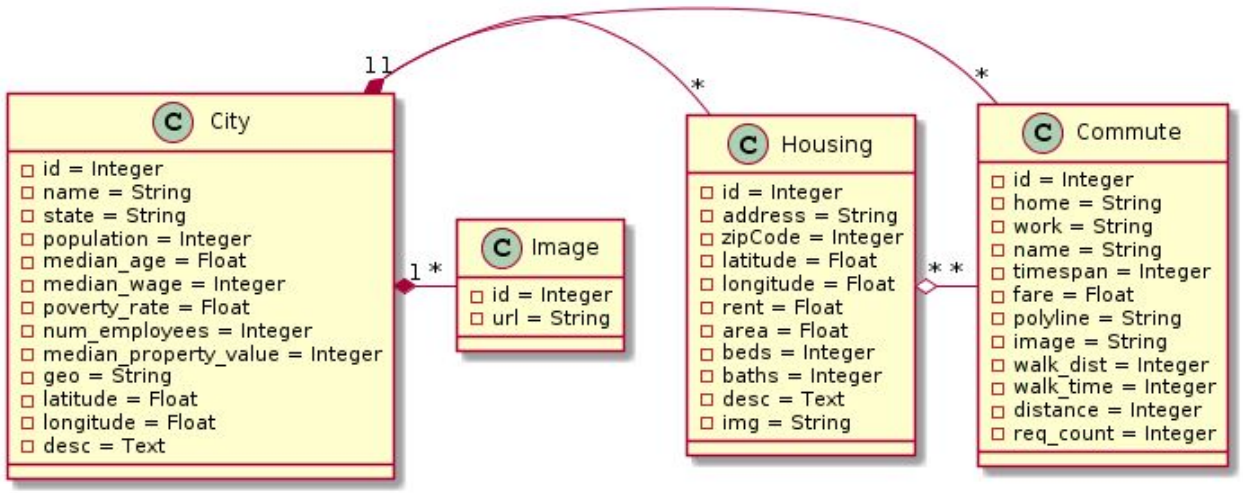
11. Add a search form to each model page - when there's a lot of data present in the database, it becomes necessary to be able to filter out results that aren't relevant to our interests. We thought it would be great to have a search form that allows the user to filter results on their own terms to find the instances they're looking for. (Estimate: < 5 hours)
12. Fix website history - Accessing the website through the browser's cached history by hitting the BACK button caused a specific error to occur. This is an important feature for a website that offers the user multiple options to compare. We'd like to support standard functionality by allowing historical views. (Estimate: < 3 hours)
13. Highlight searched text - Highlighting searched text is important for the user to know where their query exactly matched and quickly determine if the instance is something they want. (Estimate: < 4 hours)
14. Create many instances - If there aren't enough instances in the database, the usability of the data is limited to those interested in the very particular artists, venues and events in the database. We need to expand our data to create more interest. (Estimate: < 2 hours)
15. Add color to the About page - the website looks a little bland to the eye, so it may be best to spice it up with a little color to give it more personality. (Estimate: < 30 minutes)

RESTful API

Documentation on the REST API is available here. It was designed using Postman.
https://documenter.getpostman.com/view/9202355/SW18vaEc

Models

Schema



We chose the housing model first and foremost because it's what our website is built around - we can't build an application to find housing without a housing model. The transport model was also one of our first choices because the original purpose was to find available housing based on convenient bus routes to the workplace. The city model seemed to follow after we decided that we would like to know the general cost of living, which differs from city to city.

Housing

Our primary model is the housing model, representing the available residences in a preferred city for users to select from. The model contains property value; address; house area in square feet; property area in square feet; and number of beds/baths. This data is scraped from the Zillow API (https://www.zillow.com/howto/api/APIOverview.htm) for individual houses and Realtor API (https://realtor.p.rapidapi.com) for bulk requests.

- Property value - gives the user an idea of the overall cost of a home and limits certain budgets.
- Address - the location of the house in the city determines a lot about the convenience of their commute, the amount of crime in the neighborhood, how far they'll have to go to the grocery store, school, etc.
- House area - gives an idea of how much space they'll have to work with. If the user is planning to move with someone else, they'll need more space to accommodate.
- Property area - does the house have a nicely-sized yard where I can host barbecues?
- # of beds/baths - like the house area, this feature tells us more about the space inside the house and how many people it's meant for.

Public Transportation

A supporting model is the public transportation route model, which represents the available public transportation routes that connect residences and selected workplaces in a city. It includes start/end points, timespan, fare, walking distance, and request count of the route. Since Google Maps offers public transport data as a part of their pathfinding, we were going to scrape their API (https://www.googleapis.com/geolocation/v1/geolocate?key=YOUR_API_KEY) and cross-reference it with our data for evaluation with housing.

- Duration - the time it takes for the daily commute can add up over time. Route duration factors into commute convenience, and may determine how early you have to get up in the morning to go to work.
- Name - the bus ID and service that runs the route.
- Walking distance - how far do we have to walk to get to the bus stop? Do we have to walk for 45 minutes in 100+ degree weather?
- Fare - How much money will it take to get to our destination? This can add up over time just like time can.
- Polyline - The waypoints in the route that represent the legs of transport to take. Google takes this and returns an image representation of the route.

City

The final key model we use is the city model, representing the living conditions and cost of a city. It includes attributes like the cost of living, population, median housing price, median wages, and global location. The most populated cities will be scraped from the Data USA API (http://api.datausa.io/api/) and stored in our database.

- Population - how many people live in the city.
- Poverty rate - shows the wealth equality (or lack thereof) in a city.
- Median age - what are the age demographics of the city on average?
- # of employees - gives insight to the work culture surrounding a city.
- Median wages - what should I expect from a job in this city? Should I expect more or less than the average given my background?

Images

This model was introduced mainly as a way to keep a variable number of images in memory, since SQL doesn't natively support lists as a column type, and pickling Python lists gave us issues in the built-in Marshmallow schema serialization/deserialization. It's a simple model, meant only to store a URL and the object that it's tied to.

Database

The models are all linked together by location - transportation routes close to the home will be linked to a home and house locations are grouped by city. Each city also has a list of bus routes that run throughout.

Because many houses and many associated commutes can reside inside a city, both the Housing and Commute tables have a many-to-one relationship with the City table. The Housing and Commute tables, however, have a many-to-many relationship with each other because a single Housing may have many associated routes that run throughout the city associated with it, and a single Commute may be used by several Housings.

To populate the database in production, we run a series of single-use scripts that scrape our APIs for data we plan to display on the website by default. Cities are initialized first because they are the most independent of the models. After that, Housings are scraped by the existing cities in the database, and Commutes are generated lazily based on request.

Pagination

In the back-end, pagination is handled entirely by the Flask-RESTless built-in API blueprint that grants us many features for free. Each page can be accessed using an integer, specifying which page to access, assigned to the parameter "page." The number of results per page can change

at the user's request, but the higher bound on it is specified when the blueprint is initialized inside of backend/init.py.

Searching/Filtering

Mentioned in the section above are the features that come free with the default Flask-RESTless API blueprint, and among those is searching a model by a list of parameters as filters. This is implemented in Flask-RESTless and abstracted away from the user - all we do is utilize the tools we're given. To implement the user query, we use the RESTless search to query relevant parameters of the models using EQ statements to find and return all the matches we can find. The city model can search against the city name, the city's average age as a minimum or maximum, the average population as a minimum or maximum, and the average wage as a minimum or maximum. The housing model can search by city name, rent as a minimum or maximum, property area as a minimum or maximum, beds as a minimum or maximum, and baths as a minimum or maximum. The commute model can search by home address and work address since each commute is generated by a given home and work address.

Since there are not many searchable parameters in the models that are distinguishable by simple text queries, the global search bar searches against the city name associated with each model and the named city instance itself. That is, if you were to search "Phoenix, Arizona" in the search bar, all the houses linked to Phoenix, the commutes linked to Phoenix and the city of Phoenix, Arizona would come up in the search results. To do so, the global search queries each of the models from the back-end by the associated city name and displays the results it receives.

Developer Visualizations

The first visualization, as the title suggests, compares the poverty rate against the employment numbers and population for ten randomly sampled city instances. It uses the attributes from the city model poverty rate, city population and number of people employed in the city. Since our website is aimed towards those looking for available housing after they've received a job offer, this could be a useful metric to gauge the cost of living of a particular city. If the poverty rate is high while the employed-to-population ratio is also high, it could mean that the cost of living exceeds average wages or that it's difficult to make ends meet. Those anomalies with high poverty rates are cities for our users to watch out for.

The second visualization is a map of all the houses in our database plotted on the US. It uses data from the housing model, taking the latitude/longitude values of each house and plotting it on the map. This visualization gives us a visual representation of where the houses up for rent that we have in the database are distributed across the country, and can give users a better idea of the concentration of available housing in certain parts of the US versus others (i.e. New England, Coastal California, Florida, and the Dallas-Ft. Worth metropolitan area have higher concentrations of housing) and give insight into housing demand for the area.

The third visualization is a plot of median income against median rent for the same ten randomly sampled cities as the first visualization. It also draws from the city model, comparing the median rent to the median income of a city. This is a metric to display how manageable the cost of housing is in each city, comparing job income to housing cost. Looking at a city's median income and rent can also help users understand if the job offer that they received is really as high-paying as they thought it was. Compared to the rent, different cities have different equivalents for salary.

Customer Visualizations

The first visualization is a bar graph comparing the popularity of each artist according to the popularity index found in the artist model. We thought this may be a useful metric to users because more popular artists tend to tour across the world, while less popular artists may only tour in-country, so comparing the popularity indices would give users a general idea of which artists would be touring more frequently.

The second visualization is another bar graph, but shows us the total money spent each day if a hypothetical customer were going to attend every upcoming show. Artists can be enabled or disabled by clicking on their names if they are to be ignored. The info is drawn from the events associated with each artist and their average cost. Tickets can get pretty pricey and for students like us, the price is a big factor. This visualization shows us how much it would cost to go to multiple shows if we choose to.

The third and final visualization is a map of all the logged in the database happening across Texas. This pulls from the location of each event instance and maps it if it's located in the state of Texas. From the map, it's apparent that the main touring locations in Texas are Austin, Dallas and Houston. This is useful information for people that want to know which locations nearby are considered essential stops for tours.

Tools

Front-end

We're using a **React** framework with **Typescript** for the front end because it's very easy and quick to prototype using React, and Typescript manages the complexity of Javascript with object-oriented practices and makes the code easier to understand or debug. We are also using **SASS** to style our pages.

Back-end

The back-end uses a **Flask** framework, and the relational database uses **PostgreSQL** and **SQLAlchemy** to simplify the SQL queries for **Python**. **Flask-Restless** also simplifies the

creation of API endpoints with default behaviors. **Marshmallow** creates schemas for each model that can serialize and deserialize them into JSON files for simple returns.

Testing

Postman is used for testing the RESTful API endpoints, and Python functions in the backend are tested individually through the **unittest** library. **Jest** and **React's** testing tools ensure that our TypeScript is functional and **Selenium** tests the entire website end-to-end through the use of the GUI, treating it as a black box.

Amazon Web Services

AWS is used to host our production build because of its ease-of-use and many built-in tools like **Elastic Beanstalk** and **Amplify Console**, and our domain name came from **NameCheap**. Our SSL certificate was automatically provisioned through Amplify. **Route 53** manages our DNS records.

Coding Tools

NPM is used to install and update all the packages our application requires by reading from the package.json file. **TSLint** is a static analysis tool for Typescript that keeps our Typescript neat, readable, and up to spec. **Sourcetree** and **GitKraken** are used for viewing repository branches and handling merges from a simple GUI. **VS Code** is used for the actual process of merging code.

Hosting

The site is deployed on AWS Amplify static hosting. To update the public site, merge into `amplify-console`.

Group #22 Website: https://www.costlycommute.me

GitLab repo: https://gitlab.com/lukaoma/phase1