State Finder

State Finder Team: Alyiah Proctor, Emily Crabtree, Anne Nguyen, Darlyn Mendez

1. **Project Definition -**
   Career transitions, job searching, and location searching are stressful activities that one usually encounters at some stage in their lives. Our State Finder Web Application is meant to aid people in these processes by creating a tool that will provide users with information that can help their decisions. The tool will mainly be targeted to those who are looking to move to another state, college students who want to see the locations that their future career could lead them to, people who are looking for a career change, or anyone that is curious about where an occupation could allow them to live in the United States. Our State Finder Web Application attempts to increase the confidence in the career choices and location choices one may make.

   The goal of the project is to create a web application that users can rely on to find the best locations for them and their families to reside in based on their potential career income. Users should be able to input a number of career choices and family info, and the site will populate with the metro areas they can afford, as well as info on each state and further infographics regarding state costs. The team hopes to create a Web Application that is reliable, accurate, easy to use, and a great resource for our users.

   The general structure of our application will be a Python - Flask framework and Heroku to aid in the development process. Possible Python libraries that we will use include GeoPandas, Matplotlib, and Flask login. The frontend will be made with HTML, CSS, JavaScript, and Bootstrap. Additionally, the data will be stored in a SQL server database that is hosted in Azure under a student subscription.

   Our goals for the State Finder Web Application will be achieved through continuous team member collaboration on a Discord server, bi-weekly Scrum meetings, web app implementation, continuous research, troubleshooting, and great teamwork.

2. Project Requirements

The functional requirements for our project is the production of a heat map to indicate livable areas based on user input. The map should change colors indicating the livability of a specific area, according to a provided color key. Furthermore, users will be able to access state details, such as state population and average salary for the user's indicated occupation and job level. The way in which this will function is still undecided, but will either be available when the user hovers their cursor over a specific area or will automatically generate as text beneath the produced heat map.

When the web application loads, it will load to the homepage. On the top of the page, there will be a navigation bar that features the website title on the left and buttons on the right. This includes a button that redirects the user to an "About Us" page, a button that redirects the user to a "Contact Us" page, and a button that redirects the user to a "Login" page. Below the navigation bar is the website title, State Finder, with a brief description/statement for the web application.

Underneath the website title and description is a section dedicated to prompting the user to enter their household information. In this section, the user will be able to select an occupation from a dropdown menu, either by clicking on a specific selection or by typing an occupation title in the search box. They are also prompted to select their job level (entry, intermediate, or senior level, the number of working adults in the household (1 working, 1 working and 1 not working, or 2 working), and the number of children in the household (0, 1, 2, or 3), all in various dropdown menus. All of the dropdown menus must have a selected option in order for the choropleth map to generate. Then, the user will click the submit button and will be automatically redirected to the map page, which shows the choropleth map and any residual income listed greatest to least. By utilizing dropdown menus and requiring that all menus have a selection, the possibility of user input error is greatly reduced and the usability of the web application is increased.

System hardware that is required in order for the user to use our web application is a CPU, a monitor or screen to display the web application's interface, a keyboard, and a mouse. System software that is required for use of the web application is an operating system on their device, a graphics card, and a web browser. The database requirements for the web application is a connection to our SQL database. The SQL database is where all the livable wage data per area is stored, as well as the available occupations and their respective salaries. The user will not require connection to this database and nothing will be stored on their local device, but they will require an internet connection so a connection to the database can be made on the web application's backend.

Security requirements for our web application is a secure login portal that will require the user to provide a unique username and password. Each user will be given a unique ID and their login credentials, as well as their personal information required for the State Finder, will be stored in our database. The database will be secure and users will not have access to information on other users, nor will they be able to directly access the salary or livable wage data.

3. Project Specification –

The statefinder has a major focus area in data analysis. The project involves manipulating and making sense of data as well as justifying those choices and drawing insightful conclusions. Our goal with the app is to consolidate data from metro areas and create a helpful one stop shop for living decisions. There is also a specification in database management and front end user design. All data used in the app will be housed in a backend server, and our UI will need to be flexible and usable to communicate with this server.

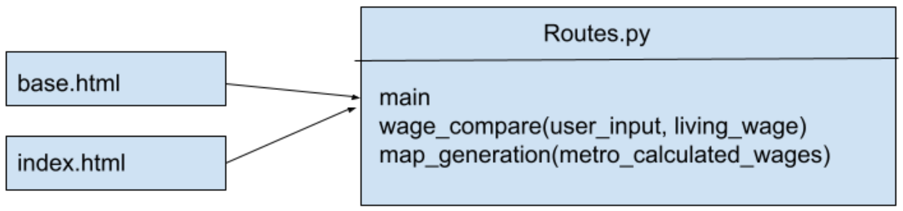
For the development of the web app the flask framework will be used to create the web aspect of the app and Azure SQL server will be the database framework for the backend. We will make changes to the Azure SQL server database in Azure Data Studio. We have all been added as collaborators to the database, so that the configuration file will work for the team for the database connection. As Python is our primary development language, the pandas, plotly, and matplotlib libraries will be used to simplify the tasks. HTML and CSS will be the primary languages used for the UI development and works hand in hand with flask. The Bootstrap front-end framework will be used to make the UI visually appealing. Any IDE is acceptable to use as long as it supports the python language. However, Visual Studio Code is the IDE used by most of the development team. We will have a Google Cloud Console Setup in order to get an API key, so that we can access the Google Maps API. We will also have a Firebase Console Setup in order to have a Firebase Realtime database to store our users login and personal information.

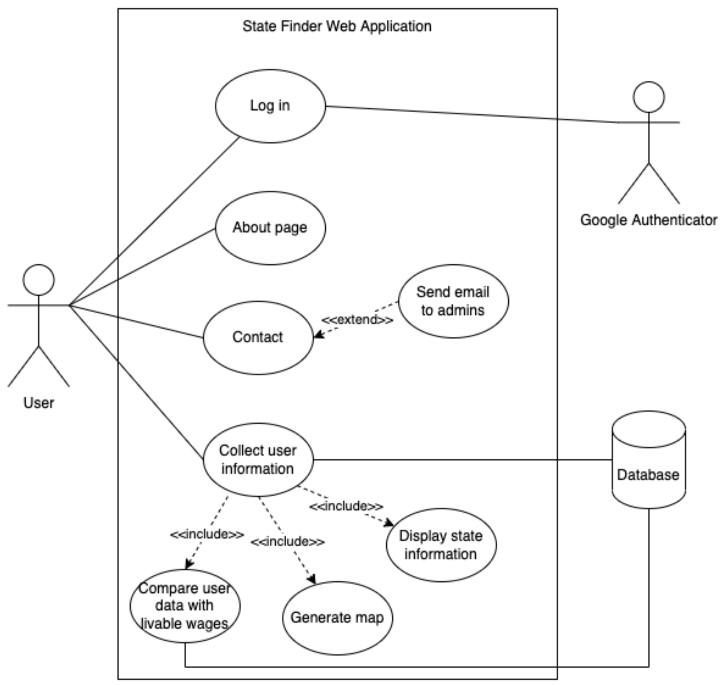
The project does not have a specific device specification. It will be built to run on most devices that can access the web through a web browser. Desktops,laptops,tablets,phones,etc.

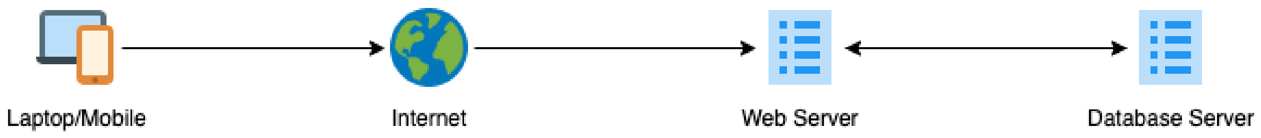
Overall the statefinder belongs in a genre of everyday web applications or sites that require very little demands from the user.

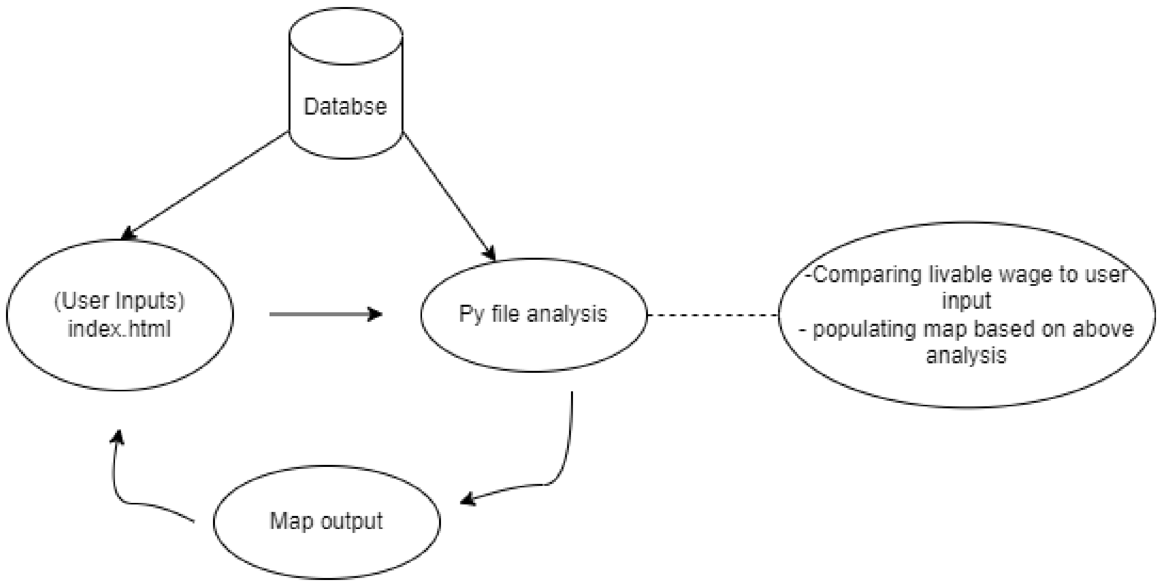
4. System – Design Perspective –

From a design perspective, our project can be broken down into three main subsystems. These subsystem are the Database subsystem, which is designed to separate livable wages by Metropolitan area, state, number of children, marital status, and whether the spouse works, and the average salaries for a large variety of occupations for ease of comparison in the code; the User Interface subsystem, which consists of the HTML files that create an aesthetically pleasing and user friendly interface; and the Python subsystem, which consists of the various Python files that are used to compare the user's information to that in the database.


base.html
index.html
Routes.py
main
wage_compare(user_input, living_wage)
map_generation(metro_calculated_wages)






Databse
(User Inputs) index.html
Py file analysis
-Comparing livable wage to user input
- populating map based on above analysis
Map output

Some design choices that we decided to implement in our project are the use of drop down menus for the user's information collection, which creates a more user friendly experience and lessens the chance of error. We also opted to produce a heat/choropleth map to display the results of the State Finder as it provides an easy to understand and general representation of where the user would be able to live in the United States based on their provided information. Lastly, we decided to use a Google authenticator for sign in to our web application, which will give the user a simpler way to sign in and negate the need for a database dedicated to storing login credentials.

As for sub-system communication, it will begin by the Python subsystem communicating with the Database subsystem to pull the data on livable wages and occupation salaries from the database into the necessary Python file. Next, the UI subsystem will communicate with the Python subsystem so the collected user information can be sent to the necessary Python file. Then, the Python subsystem will compare the user data with the Metropolitan livable wage data and the occupations salary data. Lastly, the Python subsystem will communicate with the UI subsystem to send the user's compared data back to the necessary HTML file, which will then be used to populate the choropleth map.

Overall, our system will communicate in the following way. Our main HTML form will collect the user's information and communicate that data to the main Python file. The Python file will communicate with the Azure SQL database to collect the information on livable wages in the Metropolitan areas and the provided occupation's salaries. The user's data will be compared with the data pulled from the database and then sent to the HTML file responsible for populating the choropleth map.

5. System – Analysis Perspective

From an analysis perspective the state finder has three main subsystems. One that calls to the database to retrieve all the data needed, another that compares the user input to the livable wage data, and the last subsystem that populates the map before it is passed to the html files. All of these subsystems will be their own python functions with the exception of the database call as it only needs to occur once. This adds flexibility as it will allow the site to effectively and efficiently run if the user decides to continuously change their selected inputs.

Our database consists of four tables: Livable_Wages, Occupations, state_descriptions, and metro_area_descriptions. The data dictionaries for all are shown below.

Data dictionary for tables 1 Adult, 2 adult 1 working, and 2 adults both working

Variable	Definition	Datatype
Metro Area	Name of Metropolitan Area	varchar
State	Abbreviation of State	char
0_kids_year	Annual wage for 1 Adult, 2 adults 1 working, or 2	float

	adults both working respectively with 0 kids	
1_kids_year	Annual wage for 1 Adult, 2 adults 1 working, or 2 adults both working respectively with 1 kid	float
2_kids_year	Annual wage for 1 Adult, 2 adults 1 working, or 2 adults both working respectively with 2 kids	float
2_kids_year	Annual wage for 1 Adult, 2 adults 1 working, or 2 adults both working respectively with 3 kids	float

Data dictionary for occupations

Variable	Definition	Datatype
OCC_TITLE	Title of the occupation	varchar
ANNUAL_MEAN	Average annual mean for the occupation across all states in the US	float
A_PCT10	10th percentile of the average annual mean for the occupation across all states in the US	float
A_PCT25	25th percentile of the average annual mean for the occupation across all states in the US	float
A_PCT75	75th percentile of the average annual mean for the occupation across all states in the US	float

Data dictionary for state descriptions

Variable Definition Datatype

state_name	Name of US states	varchar
state_abbreviation	Two letter abbreviations of US states	varchar
capital	Capital of US states	varchar
total_population	Population of each US state	float
total_housing _units	Number of all housing units in US states	float
occupied_housing_units	Number of occupied housing units in US states	float
vacant_housing_units	Number of vacant housing units in US states	float
percent_occupied	Percent of housing units that are occupied	float
percent_vacant	Percent of housing units that are vacant	float
attraction	An attraction that is popular in the state	varchar

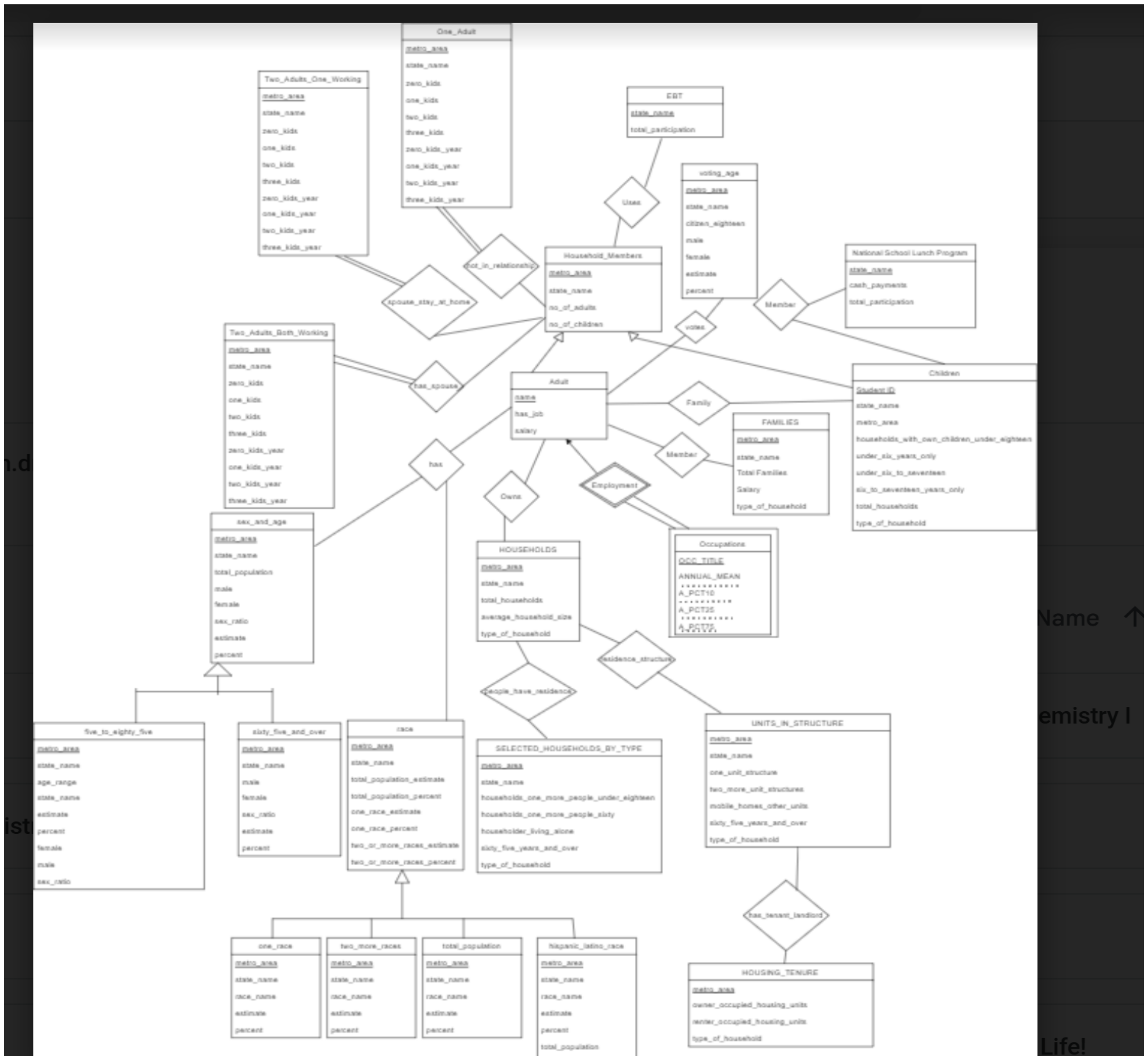
Data dictionary for metro area descriptions

Variable	Definition	Datatype
metro_area_name	Name of metro areas	varchar
state_abbreviation	Two letter abbreviations of US states	varchar
total_population	Population of each metro area	float
total_housing_units	Number of all housing units in metro areas	float
occupied_housing_units	Number of occupied housing units in metro areas	float
vacant_housing_units	Number of vacant housing units in metro areas	float

percent_occupied	Percent of housing units that are occupied	float
percent_vacant	Percent of housing units that are vacant	float

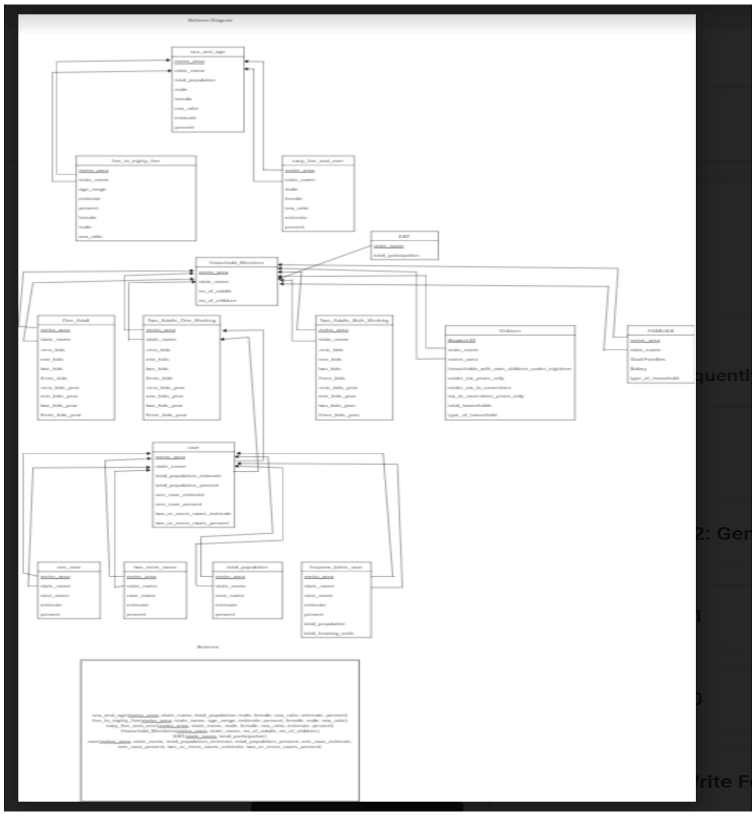
As the layout of our project is very simple, our overall system has a runtime of Big O of n. The program should only have to run through all methods once to receive the required results. If the user calls n times it will run n times.

ER Diagram



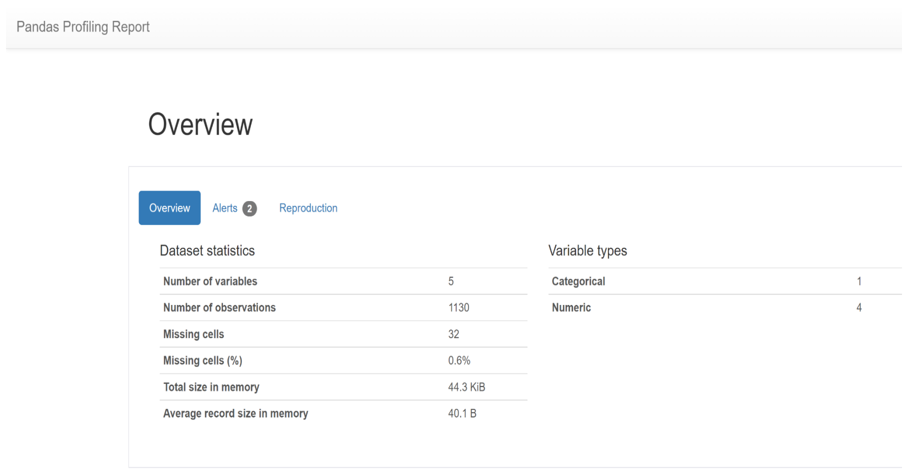
In this entity relationship diagram we tried to find any relations between the different sources of data. We identified similar columns of data through foreign key constraints in order to find connections between the datasets. One of the datasets which is the state info population dataset had to be separated into tables because the layout of the data needs to be cleaned up. Some new variables were added in order to complement the data and to reduce any confusion when writing queries for the data. Most of the tables have a relation to the Adults table which can be seen as the center of the ER diagram. Some of the tables require total participation because people have to have an age. There is disjoint generalization for race because you are part of one race or two races, but not be a part of both groups at the same time.

Schema and Schema Diagram



sex_and_age(metro_area, state_name, total_population, male, female, sex_ratio, estimate, percent)
five_to_eighty_five(metro_area, state_name, age_range, estimate, percent, female, male, sex_ratio)
sixty_five_and_over(metro_area, state_name, male, female, sex_ratio, estimate, percent)
Household_Members(metro_area, state_name, no_of_adults, no_of_children)
EBT(state_name, total_participation)
race(metro_area, state_name, total_population_estimate, total_population_percent, one_race_estimate, one_race_percent, two_or_more_races_estimate, two_or_more_races_percent)

Most of the tables have a foreign key constraint on the attributes state_name and metro_area which shows that these datasets have these columns in common. Some of the attributes are the same in multiple tables, but they are reflecting different data so foreign key constraints do not apply. The Schema and the Schema Diagram does not show us much compared to the Entity Relationship Diagram, but it helps give us a high level overview of where our foreign key constraints are. We also did some quick exploratory data analysis of some of the datasets by using Pandas Profiling to generate an HTML EDA report which can help gain an idea of where to focus our cleanup efforts on. We noticed that there are some missing values in one of the datasets and that there is a high cardinality. This means that we have to search where those missing values are coming from.


Pandas Profiling Report

Overview

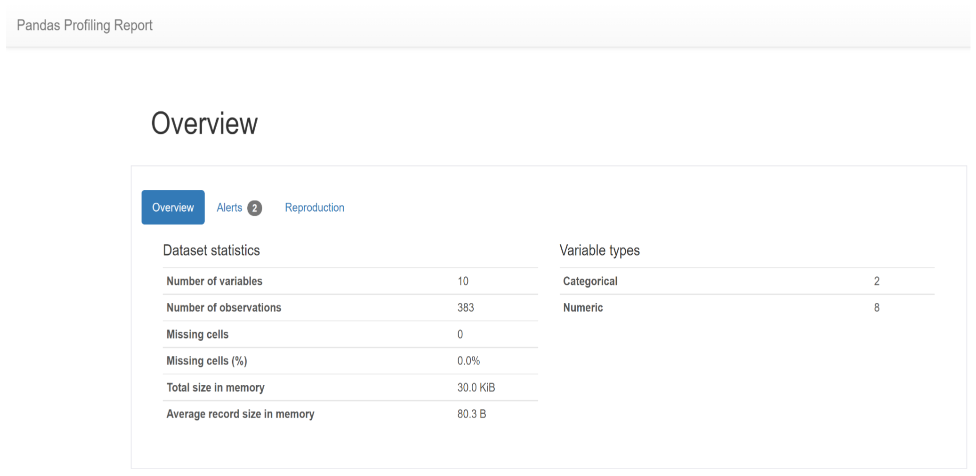
Overview | Alerts 2 | Reproduction

Dataset statistics

Number of variables	5
Number of observations	1130
Missing cells	32
Missing cells (%)	0.6%
Total size in memory	44.3 KiB
Average record size in memory	40.1 B

Variable types

Categorical	1
Numeric	4


Pandas Profiling Report

Overview

Overview | Alerts 2 | Reproduction

Dataset statistics

Number of variables	10
Number of observations	383
Missing cells	0
Missing cells (%)	0.0%
Total size in memory	30.0 KiB
Average record size in memory	80.3 B

Variable types

Categorical	2
Numeric	8

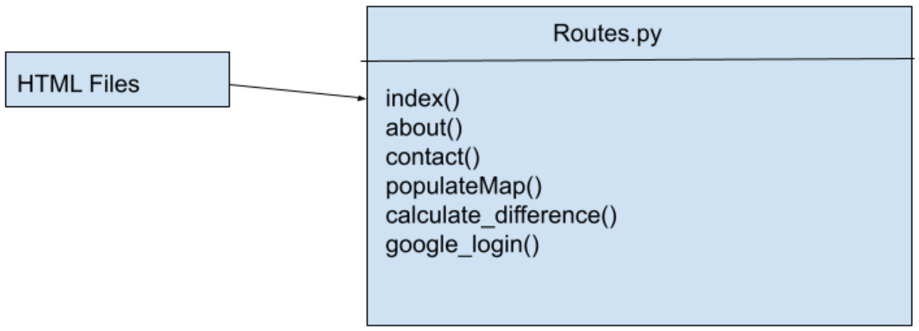
7.1 Subsystem 1 – Map Population-

This subsystem details the population of a map with metro areas based on a user's residual income. The initial plan for the included a chloropeth map with highlighted metro areas in green yellow or red depending on the users residual income if inhabiting the area. The residual income is calculated by the following equation : job income - cost of living. The initial map was to also include a hover feature to show supplementary information on each state. Each step in creating the map was to be its own function for flexibility purposes.

During implementation a few changes occurred. First the map only consists of the colors green and red. The chloropeth map does not change colors fast enough after the 0 residual income mark, meaning that if three colors were used, some states would be a dark yellow even though they should be red. The three colors provide too much variation for the range of values provided.

Secondly there is no hover feature on the residual income map. For implementation purposes, we found it easier to have state information as its own map and html pages. So that development could happen efficiently and in parallel.

Finally there are only two functions used to populate the map. The first one is the function of the html page itself, called populateMap. And this runs everytime the page is refreshed. The second is called calculate_diffierence, and this function calculates the residual income and stores it as an array with incomes in one column and metro names in another. Below is an updated class diagram with the functions. The other functions simply render the html file under the same name. Each python function has a corresponding html file.



Regarding implementation specifics, once the test page is called the user input from the home page is retrieved. This includes the job and household metrics. These values are then used to choose the correct table from the database, 1 adult, 2 adults 1 working, or 2 adults both working. The number of kids parameter from the homepage is used to extract the correct column from the table chosen, and then the metro area, state name, and livable wage values are all stored as a dataframe into a variable called metro_wage. This dataframe is passed to the calculate_difference function and a multidimensional array with the area name and residual income is returned, this is stored as variable b.

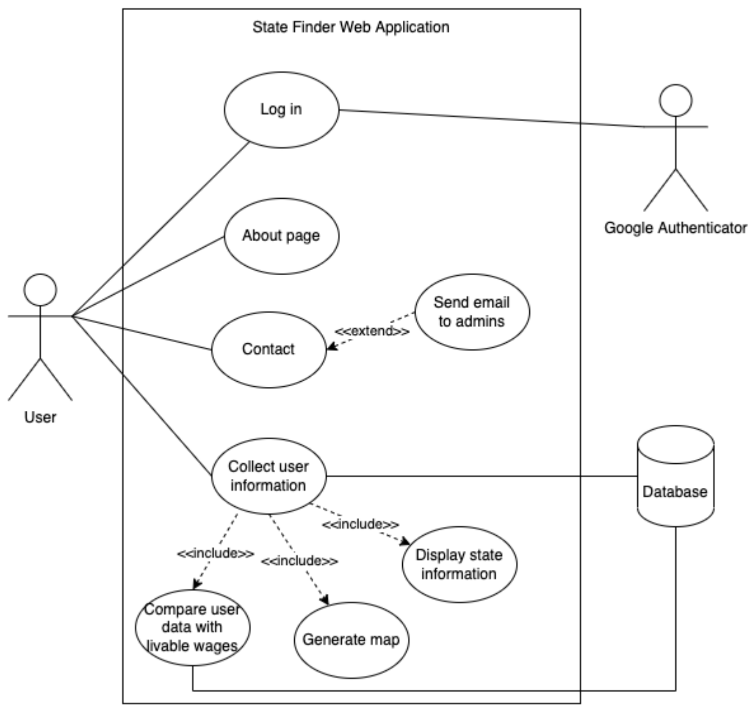
We then need to declare the shapes for the metro area. A shapefile from the us census data was used and converted into a geojsonm file. All of the metroareas from our azure database are converted to the form of metro name comma state name, in order to match the geojson names. An array goes and creates an ID, area and state name, for each column in the geojson file. This way when we plot the graph, plotly can match the json column with coordinates to out residual area columns. Then Plotly express is used to create the map. The table with residual incomes as well as the map figure are all passed to the test.html page.

Since the map highlights metro areas, a geojson shape file is used in order to provide the polygonal metro boundaries.

Testing scenarios consisted of the following, selecting a combination of random inputs on the homepage, and selecting occupations for which we lack accurate data. As user input is limitted to drop downs and only selections provided by us developers can be used, the test cases of selecting random inputs resulted in no error. However some columns of occupation wages were blank or N/A in the data retrieved from the BLS. This caused an error when producing the map as the calculate difference function did not have all of the parameters needed. To fix this a check was added in the populate map html page and in the routes.py file. If there was no data then a string that says so was passed to the html and the calculate difference function was skipped. And if there was data then a map with the livable metro areas was passed and the calculate difference function was run as normal. The check tested the datatype of the variable and printed each accordingly.

7.2 Subsystem 2 – Front end design -

Pictured below is a Use-Case Diagram for the user's interactions with the web application.



Essentially, the user will have the option to log in, visit the about page for more information about the creators and the creation process, send an email to the administrators, and input their information to populate their livable areas map. Some design choices we decided to make for the front end of the web application is to only allow the user to input information via drop down menus to minimize the chance of user error and create a more user friendly experience with less potential confusion. However, because of the size of the occupations list, there is also a search feature so that the user may begin to type the occupation and the provided list will be reduced to only ones that include the entered keyword.

Since the front end is not working directly with any databases, there is not a data dictionary.

Originally, I created a rough visual draft of the web application with an index page only devoted to welcoming the user to the website and providing links to other webpages. However, I decided that doing so left a lot of empty space, so it made more sense to put the information collection drop down menus on the index page with the welcome message. Doing so eliminated that extra space and required the use and design of one less html file. Another refinement made was to merge the about and resources pages, since neither of them would be very lengthy. Merging these two pages created a more visually appealing single page, rather than two pages with empty space.

Another refinement made, which is not pictured in either of the Use-Case diagrams, is changing the labels in the drop down menus. The drop down menu for selecting the number of children originally had the options of “0 children”, “1 child”, “2 children”, and “3+ children”. However, in hindsight, the cost of having 3 children in the household vs 10 children, which would be included in the 3+ children option, are significantly different, so I changed the “3+

children” option to just “3 children”. Other refinements included font of the website, which bounced back and forth between Lato, Roboto, and Raleway, but settled on Lato, and the color palette, which originally used #DF362D for the entire background instead of the navbar and jumbotron, and #FBAE03 for the text. The settled color palette is #DF362D for the navbar and jumbotron, #F5F0E1 for the text in the navbar and jumbotron, as well as the background color for the rest of the page, and #303030 for all text outside of the navbar and jumbotron.

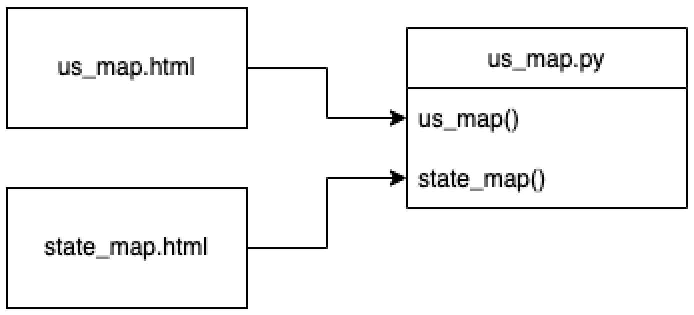
The front end was coded in HTML, CSS, Javascript, and Bootstrap, so there wasn’t any functional programming. All of the object oriented programming is done with Javascript and helps the page scroll smoother. It will also be used with the implementation of the contact page, which is currently still being developed.

From the index page, the user will be prompted to select their occupation, job level, whether there is 1 working adult, 1 working and 1 non-working adult, or 2 working adults in the house, and the number of children in the house up to 3. Then, the user will click the submit button and be redirected to the test.html page where the populated choropleth map will be displayed, along with the metropolitan areas and excess wage listed from highest to lowest. If the user does not enter their information, there are a variety of other pages they can visit that are listed in the navbar at the top of the page. They can visit the about page, which gives information of the creators of the web application, the creative process/purpose behind it, and the resources and references used in the creation of the web app. They can also visit the contact page, which allows the user to send an email to the administrators of the web application in case of an issue or question. Finally, there is a link to log in to the application.

Testing for front end development was done by making changes to the code, saving the files, and running the flask application as a localhost to see how the changes visually look. We also made sure to check all instances where a change in the code would take place, instead of the specific instance in mind when the change was made. For example, finding everywhere a CSS selector is used after a change is made to ensure that the changes do not negatively impact the visuals or function of the webpage. Any buttons created for the purpose of redirecting the user to another webpage were thoroughly checked to ensure that no errors occurred and links were input into the code correctly. All web pages that performed some other purpose other than basic functionality, such as the contact page, were manually tested to guarantee that they functioned as expected without error.

7.3 Subsystem 3 – State infographics-

The general design and model for this subsystem is as follows:



The specific data dictionaries that are used in this subsystem are state descriptions and metro area descriptions. The state descriptions data is used when displaying the states of the United States and the metro area descriptions data is used when displaying the metro areas of a state that the user selects.

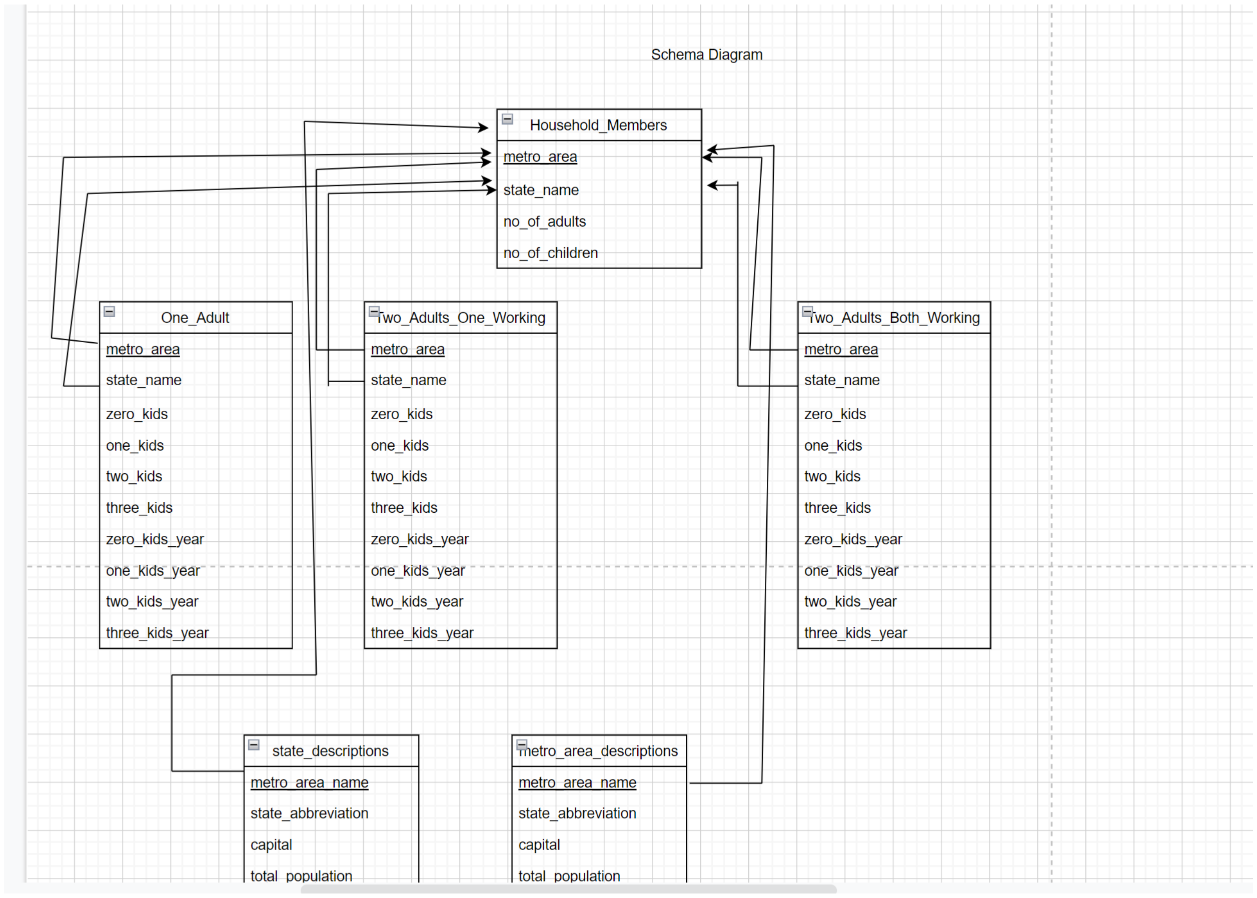
The selected state info page changed from a detailed view of the state consisting of a brief description and any additional graphs to a map of the metro areas of the state with the ability to hover over the metro areas for additional information similar to the US map. Since our livable wage calculations are based on the metro areas, providing users with population and housing statistics of the metro areas would be more beneficial than providing them with more detailed information about the states.

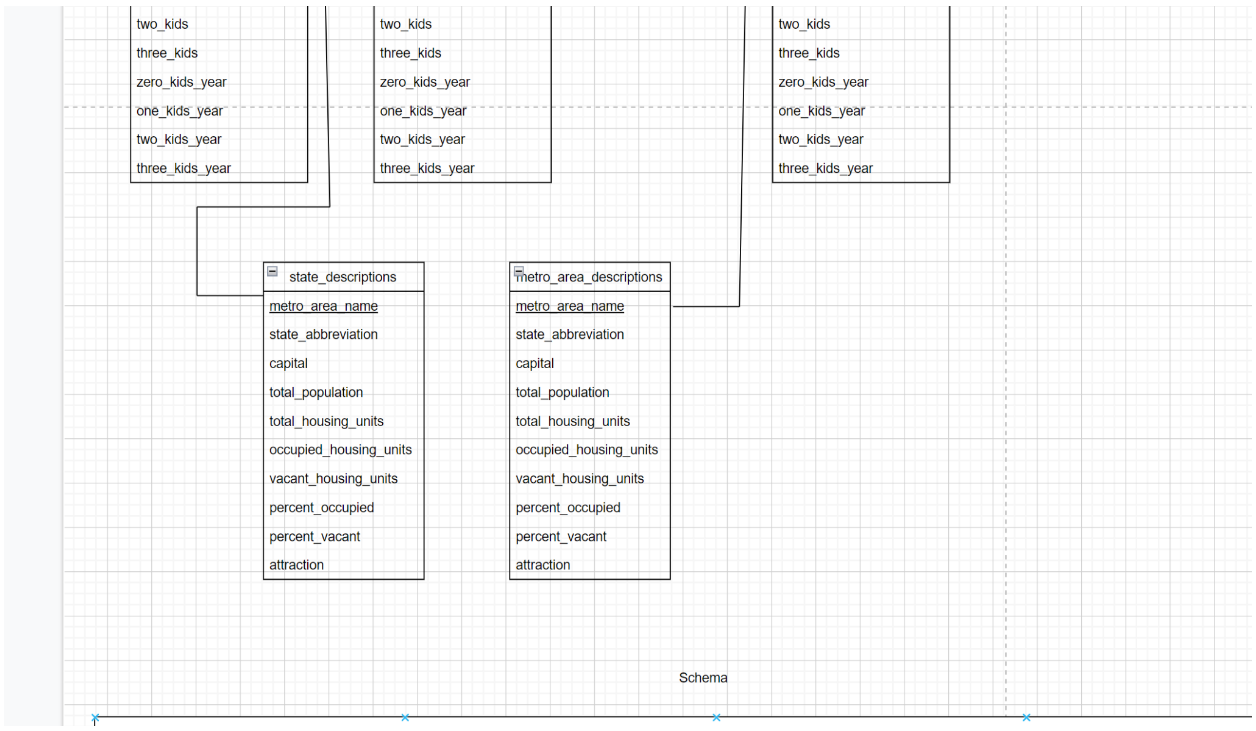
This subsystem was coded using mainly the choropleth map feature of Plotly Graph Objects. The choropleth scale of the maps is based on population size. The ability to view information about specific regions by hovering over them on a map was also implemented using Plotly. Information about the states and their metro areas is called from the database. In this subsystem, there are two functions for map generation. The first function us_map() returns an interactive US choropleth map to us_map.html. The second function state_map() returns an interactive state choropleth map to state_map.html. When the user selects a state from the drop-down menu, the value is returned to this function and is used to fix the scope of the map to the selected state. Then, by using a geojson file from the US Census Bureau, the metro areas of the selected state will be able to be displayed. The language used in this subsystem is Python.

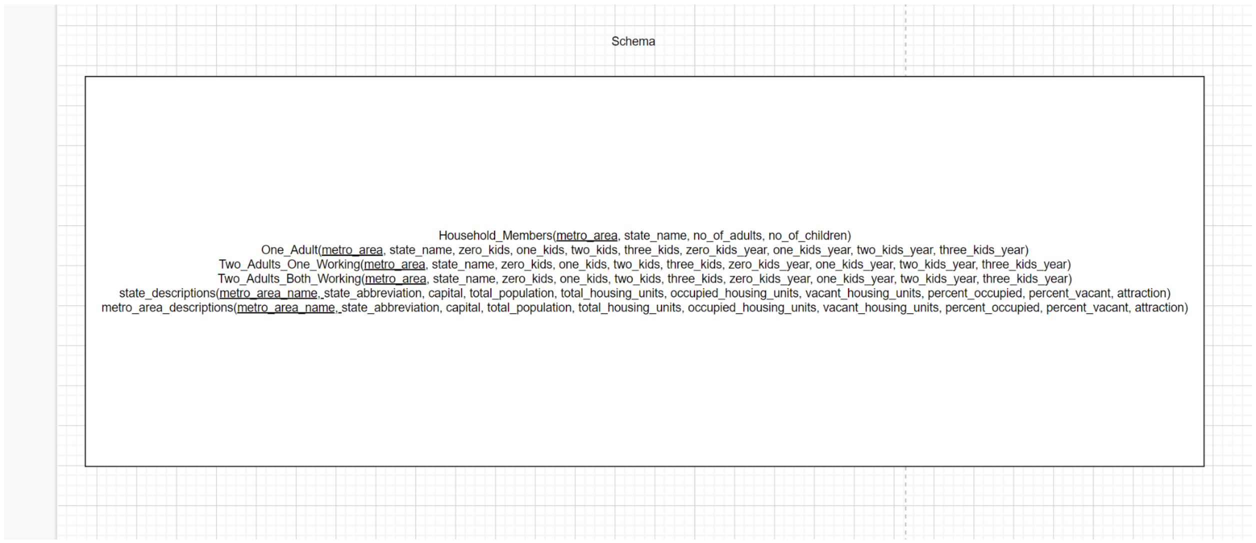
The user will navigate to the state infographics page and from there they will be able to view a US choropleth map based on population size. The user will be able to view information about the states by hovering over the map. The type of information that is displayed for each state is the state name, capital, total population, total housing units, number of occupied housing units and vacant housing units, the percentages of occupied and vacant housing units, and a popular attraction within that state. The user will also be able to select a specific state from a drop-down menu. Then, they will be taken to another page that shows a choropleth map of the selected state based on population size. The user will be able to view information about its metro areas by hovering over the map. The type of information that is displayed for each metro area is the metro area name, total population, total housing units, number of occupied housing units and vacant housing units, and the percentages of occupied and vacant housing units. The user will be able to return to the US map by clicking the “Back to US” button and select a different state if desired.

For the testing phase, the data of the states were tested to see if they matched with the states on the US map and the data of the metro areas were tested to see if they matched with the metro areas on the selected state map. The state the user selects was also tested to see if it matches with the state that appears on the selected state map.

7.4 Subsystem 4 – Database-







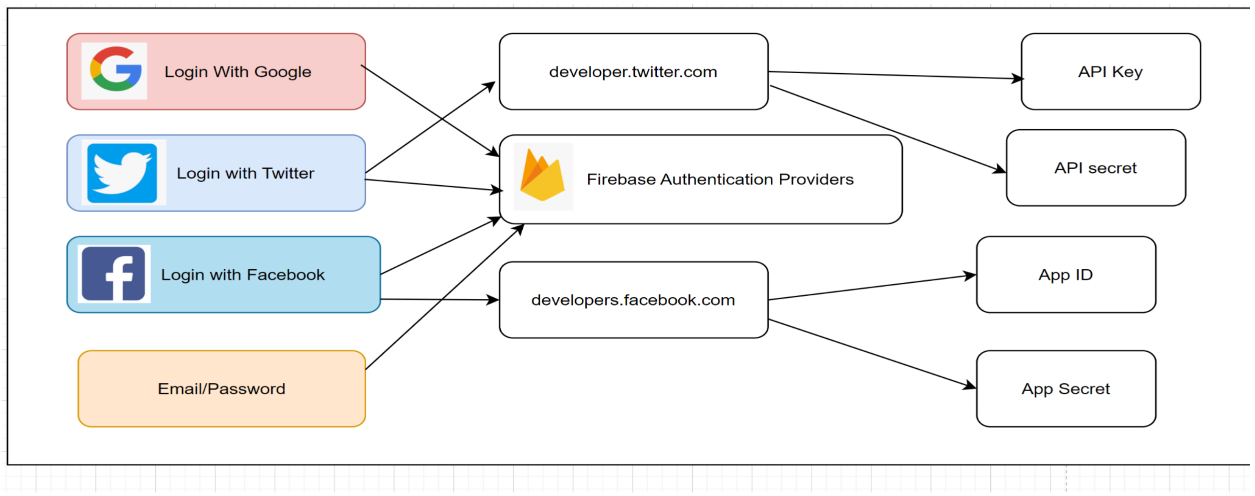
Schema

Household_Members(<u>metro_area</u>, state_name, no_of_adults, no_of_children)
One_Adult(<u>metro_area</u>, state_name, zero_kids, one_kids, two_kids, three_kids, zero_kids_year, one_kids_year, two_kids_year, three_kids_year)
Two_Adults_One_Working(<u>metro_area</u>, state_name, zero_kids, one_kids, two_kids, three_kids, zero_kids_year, one_kids_year, two_kids_year, three_kids_year)
Two_Adults_Both_Working(<u>metro_area</u>, state_name, zero_kids, one_kids, two_kids, three_kids, zero_kids_year, one_kids_year, two_kids_year, three_kids_year)
state_descriptions(<u>metro_area_name,</u> state_abbreviation, capital, total_population, total_housing_units, occupied_housing_units, vacant_housing_units, percent_occupied, percent_vacant, attraction)
metro_area_descriptions(<u>metro_area_name,</u> state_abbreviation, capital, total_population, total_housing_units, occupied_housing_units, vacant_housing_units, percent_occupied, percent_vacant, attraction)

The initial design and model of the database includes an ER Diagram and a Schema Diagram that would help us restructure the csv files that we have into tables that we can use later on. The Design Choices that we made is to separate two of the data files into several entities to reduce confusion when we reference them during implementation. We used diagrams.net and its UML shapes to create the ER diagram and the schema diagram. We tried our best to make the notation accurate with the shapes that are available. The data dictionary includes attributes from these entities and their data types. The changes from the initial model include the addition of the state_descriptions entity and the metro_area_descriptions table. These tables were added so that we can have a detailed view of each of the states in the UI. The pros of the refined design is that the user is going to be able to use this new feature.

We used an Azure database. We connected to this Azure database using Azure Data Studio. Each of the members was given permission to create changes in the database. The Azure database utilizes SQL Server for any changes to be made to the database. We saved all of the code for the database in a .sql file that we pushed to our Github repository. Our approach with the database is that we used the primary keys and the foreign key constraints to connect the database tables with each other. One of the issues that we faced is the foreign key constraint errors because there were duplicate values, so we matched the values from the two different tables in order to find the values that were causing this issue. The database has been tested through its connection to the flask web app. We have to make sure that we re authenticate ourselves. We have an error with the sql server driver when running the flask web app. One of our members' Azure credits was disabled so we had to give the database permission to access our user IP address.

7.5 Subsystem 5 – Login-



This diagram is showing what we provide to the project in our firebase console from the developer sites. We are mostly providing the API key and the API secret to the project in our firebase console so that our new providers can be successfully added. The Email/Password provider is going to be used by our team members to resolve any issues that users may be facing. The users of our web app are going to be encouraged to authenticate themselves through one of our providers for the best experience possible. Used firebase docs as a reference for the implementation of the login subsystem. We included service provider icons in the UI, so that users can easily identify the service provider that they would like to authenticate themselves with. We initially were just focusing on login through google authentication, but we wanted users to have other options as well. For the implementation of this subsystem we used Javascript.

When we created the project in our Firebase console they provided some javascript sdks so that we could connect to the firebase web app in our google_login.html file. We had to make sure to include the link to our firebase app to connect the project on developers.facebook.com to the firebase console. We tested it on localhost, so it should work once we deploy it and are able to give permission to that domain on the firebase console.

8. Complete System

Heroku was used as the deployment tool for the site. The site can be accessed at https://statefinder.herokuapp.com/

Deployment was done using the heroku website and linking the site to a github repository. This process included adding a Procfile to communicate with Heroku as well as a requirements.txt file. The github with the code can be accessed here.

Team Member descriptions and responsibilities

- Alyiah Proctor
    - Livable Areas map and table generation
    - App deployment

- Anne Nguyen
    - Metro area infographic data search and subsystem generation

- Emily Crabtree
    - Home page
    - All HTML subsystems and info pages
    - App deployment

- Darlyn Mendez
    - Database configuration
    - Login configuration