Technical Report

Team 22: Travel Scares Me

Table of Contents

Motivation

In the current era of COVID-19, travel has become very unsafe and has been discouraged if it is not necessary. For those who need to travel, we aim to bring the latest information about airplane routes between different countries, what travel advisories are currently imposed on these countries, and the latest news articles about these countries. We also provide the latest COVID-19 statistical data and people's behavioral data to make travel decisions easier.

Data Models

Countries

Our first model is the Country model, which contains general information about a country and data about its current COVID-19 situation, along with travel advisories. This model also connects to our two other models (see below), as news articles originate from a country and the airplane routes connect two countries.

- Filterable Attributes: Latitude, Longitude, Region, Population
- Sortable Attributes: Name, Country Code, Latitude, Longitude, Capital, Region, Population
- Additional Searchable Attributes: Currencies, Languages, Neighboring Countries, Timezones, COVID-19 Data, Threat Level
- Media: Country Flag, Country Map, COVID Data Pie Charts

Airplane Routes

Our second data model is the Airplane Routes model. This model contains flight route information, including the airline and flight number, source, and destination of the flight. This model directly connects to the country model via the source and destination country, and the news articles about the destination country.

- Filterable Attributes: Airline Code, Flight Number, Departure Country, Departure Airport, Arrival Country, Arrival Airport
- Sortable Attributes: Airline Code + Flight Number, Departure Country, Departure Airport, Arrival Country, Arrival Airport
- Additional Searchable Attributes: Airline Name, Departure (Country, City, Airport, Code, Time Zone, Time, Terminal), Arrival (Country, City, Airport, Code, Time Zone, Time, Terminal)
- Media: Departure Country Flag, Departure Airport Map Location, Arrival Country Flag, Arrival Airport Map Location

Articles

Our third model is the Articles model. This model contains a list of news articles from around the world. Each article has an author, source, published date, image, and country associated with it. There is also a link to the actual article in case the user wishes to access the source. These news articles connect back to the country of origin, and the airplane routes that serve that country.

- Filterable Attributes: Category, Country, Language
- Sortable Attributes: Headline, Category, Published Date, Country, Language
- Additional Searchable Attributes: Authors, Source, Description, Source Description, Source Website
- Media: Image of the article, Image of source logo, Relevant video

Data Collection

Data was collected from various sources for the different models. The data was collected using the requests library in Python and was formatted into the required format using the json library in Python. Also, the data was joined together in this stage itself, for example, by using the countries data to fill in the country fields for airplane routes and news articles.

Countries

The generic data for countries, the COVID data, the human behavior data, and travel advisories data were all collected from different sources and combined into a single data model that can be requested by the frontend. The sources are as follows:

- Generic data: https://restcountries.eu/
- Travel advisories and threat level data: https://www.state.gov/developer/
- COVID cases data: https://api.covid19api.com/
- Human behavior during COVID data: https://covidmap.umd.edu/api.html

Airplane Routes

The airplane routes data was all collected from a single source although a lot of different requests were made to combine data into a human-comprehensible format. The routes themselves returned IATA codes for airports, airlines, cities, and countries. Requests were made to all these entities to retrieve proper names and then combine the data. The source used is https://aviation-edge.com/.

Articles

The news articles' data itself was collected from a single source. However, the data collected was not very rich in attributes. This same source also provided data about different news sources. The news articles' data was combined with the sources' data. Additionally, a search API was used to retrieve images of the news sources' logos. The sources are as follows:

- Articles and sources: https://newsapi.org/docs/
- Search API for images: https://serpapi.com/images-results

Dynamic data

Other data such as the maps and videos displayed on the instances pages for all the models, and commits, issues data on the about page are collected dynamically when the page is being rendered and are not stored in the database. This data is retrieved from the following sources:

- Maps: https://developers.google.com/maps/documentation/embed/get-started
- Relevant YouTube videos: https://developers.google.com/youtube/v3
- Gitlab commits, issues: https://docs.gitlab.com/ee/api/README.html

Database

When setting up the database, the main design decisions to be made were how to model the one-to-one, one-to-many, and many-to-many relationships. Accordingly, some of the parts of our models were broken into separate tables.

Countries

Each country model potentially has several languages, currencies, time zones, and neighbors. These four properties of a country were separated into separate tables. The relation between countries and any of these tables is many-to-many (e.g. English is spoken in the USA and Canada, but Canada also speaks French). To link them back with countries, there is a need for junction tables that just have the IDs of the countries table and the other table as foreign keys.

- countries
- languages (junction: countries_languages)
- currencies (junction: countries_currencies)
- neighbors (junction: countries_neighbors)
- time_zones (junction: countries_time_zones)

Airplane Routes

Each airplane route model has two main components: information about the departure airport and information about the arrival airport. This information was separated into an airports table and foreign keys to the airports table were placed in the routes table. Additionally, the airports table has a foreign key to the countries table to allow for retrieving the country ID to display on the website. Lastly, the routes table has indexes for the departure and arrival country names to make filtering airplane routes easier in countries and articles.

- airplane_routes
- airports

Articles

Each article model is complete on its own. The only dependency it has is a foreign key to the countries table to allow for retrieving the country ID to display on the website. No special index was required for the news_articles table because the filtering of articles was done based on the country ID which is a foreign key and automatically indexed.

- news_articles

RESTful API

Postman documentation: https://documenter.getpostman.com/view/14734911/Tz5jeL2c

We designed two sets of APIs. Each of our models has two API routes associated: a GET for a single instance of the model, and a GET for listing several instances of the model.

GET single instance

The single instance APIs return a single object of an instance of a model. The ID of the instance is required to use these endpoints. These endpoints return all the data corresponding to the instance, including the data that requires joining database tables. However, relevant airplane routes, and relevant articles were implemented as filters instead of attributes in the database to avoid joining the large tables with a lot of attributes together and slowing down the response time.

The following are the single instance API routes:

- https://travelscares.me/api/countries/:id
- https://travelscares.me/api/routes/:id
- https://travelscares.me/api/articles/:id

GET list of instances

The list of instances APIs all return the total number of instances available for the model and an array of the requested number of results. Because of the nature of our data models having a really large amount of information (such as country travel advisories), we only return the relevant attributes to display on the model pages.

The required parameters for all list of instances APIs are page and pageSize to determine the amount of results to be returned and which results need to be returned. These APIs also support filtering and sorting on certain attributes, and also allow for searching on all the attributes of each model.

Each API has its own parameters for filtering that are named similarly to the attributes they are filtering. Multiple filters can be applied together but only one value can be applied for a single filter. The filters for numeric values (like country population) are split into two attributes (populationMin and populationMax in the example) to make it more explicit when using the API.

All the APIs have common parameters for sorting. The orderBy and ordering parameters are used to determine sorting by which attribute and in which direction (ascending or descending). Either both the parameters have to be present or neither should be present.

All the APIs have a common parameter for making a search query. The q parameter is used to

determine the search query across all displayable attributes of an instance of a model. This includes attributes that are not necessarily visible in the preview of the instance on the model page. This query can be as long as needed.

The following are the list of instances API routes:

- Countries: https://travelscares.me/api/countries?region={string}&latitudeMin={int}&latitudeMax={int}&longitudeMin={int}&longitudeMax={int}&populationMin={int}&populationMax={int}&orderBy={name|code|region|population|capital|latitude|longitude}&ordering={asc|desc}&page={int}&pageSize={int}&q={string}
- Airplane routes: https://travelscares.me/api/routes?airline={string}&departureCity={string}&arrivalCity={string}&departureCountry={string}&arrivalCountry={string}&orderBy={flight|departure_city|arrival_city|departure_country|arrival_country}&ordering={asc|desc}&page={int}&pageSize={int}&q={string}
- News articles: https://travelscares.me/api/articles?category={string}&country={string}&language={string}&orderBy={category|country|language|title|published}&ordering={asc|desc}&page={int}&pageSize={int}&q={string}

Pagination

Frontend

The frontend handles the current page and page size through state management. To change the state, we have buttons to dispatch events according to what the user wants to do. For example, at the top and bottom of each model page, there are buttons to go to the first page, previous page, next page, and last page. When these buttons are clicked, a mouse event is fired and captured by a handler that updates the current page of the user. Once the current page is updated, the parameters of the query to the backend is updated, so the query executes with its new parameters to get the new page and display its results on the model page. The current page can also be changed through the dropdown at the bottom of the page to select a specific page the user wants to go to. Additionally, there are bounds checking for the pagination buttons to disable them when necessary. For example, when the user is on the first page, the first page button and the previous page button will not be able to be clicked. Similarly, when the user is on the last page, the next page and last page button will not be able to be clicked.

Similarly, the page size can be changed through a dropdown at the bottom of the page. When the page size is changed, the current page becomes the first page, and the page is loaded with the number of instances denoted by the new page size.

Backend

Pagination is mostly handled on the backend through the API request. Based on the page number and the page size in the API request, the appropriate results are extracted from the database and packed into the response. This is done in one of two ways based on the model for which pagination is being handled.

For countries and articles, the pagination is done using SQLAlchemy's automatic pagination using the paginate() function that uses the page number and page size and returns a special Paginate object that cannot be further queried.

For airplane routes, the airplane_routes table needs to perform a join with the airports table. This would be costly on the entire airplane_routes table. After the join is performed, the pagination is done in the same way as for countries and articles.

Filtering

Frontend

Filtering on the frontend handles the current state of the filters being applied. There is a JSON that holds the initial state of the filters: the type of filter, active states set to false, the values they can hold set to empty strings, the options available for dropdown filters, the extents of a range for min and max filters, and the name of each filter.

Once these are initialized, they can be modified by opening the filter modal through the "Change Filters" button. The modal initially shows all the filters in empty states, but they can be modified through interacting with them. The select filter is a dropdown with a text input inside it. It allows the user to either scroll through the dropdown menu to find their filter, or search for one through the text input. Options are filtered by the search query by matching each option that started with the search query. For range options, there are min and max number inputs. Each can be filled out individually, but if the user tries to apply a min greater than a max, an error message will show indicating this mistake.

To apply the changes, the user will need to click the "Apply" button, which will exit the filter modal, and show the filters applied as badges next to the "Change Filters" button, and through the results with a query to the backend serializing the filters as query parameters in the URL. If the user exits the modal without applying changes, the changes will be removed. If the user wants to clear a certain filter, they can do so by clicking the trash can button next to the filter input.

Backend

Filtering on the data is completely handled in the backend through the API request, no filtering is performed in the frontend. Based on the selected parameters (could be multiple) in the API request, appropriate results are extracted in database insertion order through SQLAlchemy’s filter() interface.

For articles, the filtering is straightforward, just need to match the parameters (region, country, language) with the appropriate columns in the database.

For countries, some filtering attributes (population, latitude, longitude) are numerical and have two parameters (a min and a max parameter). SQLAlchemy can still easily handle this by providing a column filter that considers a min limit and/or a max limit. The remaining parameter (region) is just matching the parameter to the column.

For airplane routes, filtering is a little more complicated because of the airplane_routes and airports tables being joined. Additionally, there are two airports that are joined with a single

airplane route requiring aliased tables to work with. The filters are applied through the aliased airports tables because most of the filtering parameters (departureCity, departureCountry, arrivalCity, arrivalCountry) are associated with columns present in the airports tables and not the airplane_routes table. The remaining parameter (airline) is just matching the parameter to the column in the airplane_routes table.

Sorting

Frontend

Sorting is maintained through states that monitor the current sort column and the current sort order. On the articles and countries model pages, sorting is done through a dropdown and a sorting order toggle. When the dropdown is selected, all the sort columns that are available are listed, and selecting one will change the sort column that the sorting is done through. The sort order is changed through the up and down arrow toggle, with up indicating ascending order and down indicating descending order. For the airplane routes page, sorting is done by clicking on the row header cells of the table. When the user clicks on a cell that is already selected, the sort order is toggled, but if a new cell is clicked on, then the sort column is changed and the sort order is set to the default ascending order. Once the sort columns and sort order are applied, they are serialized along with the filters as query params in the URL to send to the backend to receive the corresponding results.

Backend

Sorting on the data is completely handled in the backend through the API request, no sorting is performed in the frontend. Sorting is always by a single attribute and in either ascending or descending order. All results are extracted in the appropriate order through use of SQLAlchemy's order_by() interface.

For articles and countries, sorting is straightforward, just need to match the orderBy parameter to the column that is being ordered. The ordering parameter is used to determine if a desc() function needs to be applied to the column.

For airplane routes, sorting is a little more complicated because of the airplane_routes and airports tables being joined. Additionally, there are two airports that are joined with a single airplane route requiring aliased tables to work with. The sorting is applied through the aliased airports tables because most of the values orderBy can take (departure_city, departure_country, arrival_city, arrival_country) are associated with columns present in the airports tables and not the airplane_routes table. The remaining possible value of orderBy (airline) is a conjunctive sorting over two columns in the database, first airline_code, second flight_num, and is applied accordingly. The ordering parameter is used to determine if a desc() function needs to be applied to the column.

Searching

Frontend

Search functionality is implemented on two levels in the frontend: on a model page, and site-wide. In both cases, react hooks are used to keep track of the search query being entered into the search bar. When the user presses enter or clicks the search button, the API parameter is updated and the search results are retrieved to be displayed.

Search results are filtered as explained in the Backend section below, implying that some attributes that matched the search criteria may not be visible immediately on the model page or global search page. However the visible terms that do match are highlighted yellow using the react-highlight-words library.

On the model pages, the search terms are highlighted yellow on the cards (for articles and countries) and in the table rows (for airplane routes). There is no other difference in how results are displayed when searching in comparison to when not searching.

For the global search, a minimized version of each model page is displayed (only contains the pagination in addition to the data being displayed). The minimized versions display lesser instances at a time than the main model pages. Each minimized version of the model pages handles its respective API calls and there is no centralized search API endpoint. The search terms are highlighted yellow on the minimized model pages in the same way as on the main model pages.

Backend

Searching on the data is completely handled in the backend through the API request, no searching is performed in the frontend. Results are extracted in database insertion order by SQLAlchemy's filter() and contains() interfaces.

For articles and countries, searching is done on each entire instance as already joined internally through SQLAlchemy's model relationships. This allows access to other attributes of the models that are not returned in the response itself.

For airplane routes, searching is a little more complicated because of the airplane_routes and airports tables being joined manually. Additionally, there are two airports that are joined with a single airplane route requiring aliased tables to work with. The searching is applied through the airplane_routes table and the aliased airports tables because some attributes are available in the airplane_routes table while the others come from the airports table.

For all models, almost all displayable attributes are searched over by checking if each token is

contained in the attribute's value, however attributes such as URLs which are not directly displayed are not searched over.

When implementing the general or "google-like" search in the backend, we added a new key parameter "q". Anything passed into this parameter would be passed into sql_alchemy's or_ function over all of our search parameters, which would in effect search through all of our search parameters for something containing that string. For example, calling api/routes?page=1&pageSize=10&q=New%20York would tell the backend to search for New York in all the search parameters of our routes API.

Visualizations

Visualizations for both our API and our provider's (Aid All Around) API are done with react-google-charts which is a React wrapper for Google Charts. This library provides a wide variety of interactable charts including column charts, pie charts, and bubble charts.

The data for a visualization is fetched dynamically from an API endpoint and is processed into the format required by the specific chart. The data is loaded only when the specific visualization tab is clicked, and is not loaded on subsequent tab clicks to prevent unnecessary reloading of the same data.

Tools Utilized

Gitlab

We use Gitlab as our source control for this project, as well as our issue tracker. Gitlab is also used for Continuous Integration to run the four different types of tests - Javascript (frontend), Python (backend), GUI, and Postman. We have sorted our repository into a frontend and backend folder, each of which contains the code for that respective portion of the project.

Postman

Our API documentation is in Postman, which can be accessed from our About page on the website. Currently, this API documentation also has functionality that has not yet been implemented but the examples show what has been implemented so far. Postman is also used as a testing framework for our APIs.

React

We use React to create our front-end website, which allows us to create web pages easily when given a JSON object. This JSON object comes from our API that is served by our Flask backend. The two main React libraries used are React Bootstrap and React Router.

React Bootstrap

React Bootstrap serves as our style framework, which we use to design and stylize our web pages. Using .jsx files, we create pages that vary dynamically based on the input JSON. This allows us to format our pages dynamically while keeping the same style throughout. React Bootstrap also gives us some premade components to work with, making it easier to make tables or grids on our model pages.

React Router

React Router serves as our navigational component for our website. This allows our web pages to navigate between each other easily and share information about what page was clicked. We use Link from React Router to create and navigate between our URLs within our .jsx files.

Flask

We use Flask to set up a web framework with routes for the web application. Flask is a minimal framework that does not have any application features. Other libraries are used in conjunction with Flask to perform database operations as expanded below.

SQLAlchemy & Marshmallow

SQLAlchemy is a library utilized for setting up database schemas and performing database operations. The database was initially populated using SQLAlchemy and is also used for accessing the database when a request is received by Flask. Marshmallow is a library utilized for setting up JSON schemas for the responses to the requests received. It takes advantage of the database relationships set up by SQLAlchemy to populate the response JSON object.

Testing

Javascript (frontend)

The JavaScript tests use the Jest testing framework along with the react testing library. Many of these tests use mock data and test whether this data actually appears on the page. This includes finding elements that contain the data, images that have the correct source URL, and checking if links point to the correct pages.

In addition to mocking the data received from network calls, modules and functions that use network calls also need to be mocked. For fetch statements, mocking included injecting the mock data created in the resolve response from the json function given from the resolve response from the fetch statement. For useQuery hook responses, the isLoading state and the data response are mocked to test certain states with each component that uses the useQuery hook.

For rendering internal links, a router wrapper around the component renderer function given by the react testing library is created to keep the same functionality of react-router links.

For filtering, sorting, and searching, the model pages are validated through interacting with each of the filter, sorting, and searching components, and checking to see if a request was correctly made to the backend.

GUI (frontend)

The GUI tests for the front end are written using splinter, a Python wrapper for selenium. They are written using Python's unittest class framework. These tests test the functionality of the front-end elements as they appear on the screen. It tests links on each of the individual pages to ensure they go to the appropriate corresponding page. It also ensures that the buttons for pagination and the several drop-down menus behave as expected.

For most of our GUI tests, they execute an operation (such as going to the next page of a model view) and then ensure the resulting page is the correct one. This is done by ensuring that the correct text appears on that page. For example, if the goal is to test the functionality of the next page button on the Countries page, the program tests to see that the correct countries appear before and after the button press. To test the Filtering and Searching parameters, the tests enter text into the field and submit the results, and then test to see if the correct text is on the page. There is at least one acceptance test per functionality on each page, and each test begins with a fresh reload of the page to prevent cross-test dependency.

The tests can be run on different operating systems by selecting the appropriate browser driver for the distribution of the computer (Linux or Windows). The tests can also be run with the

deployed version of the website or a localhost version of the website. All these selections can be made by modifying Global.py in frontend/src/tests/gui. Different operating system Chrome chromedrivers and Firefox geckodrivers are both stored in the repository, and can be used for testing. Currently, we use Chrome for our testing and CI pipeline. We have not tested Firefox through our CI pipeline due to setup difficulties.

Python (backend)

The tests for Python are done using the unittest class framework. For each of our single instance APIs, we create a request to the API for a single instance of that model. We test for the presence and type of each of the specific attributes of that class. By extracting the json response from the API requests, we test for the presence of attributes like “capital” and sub-attributes like “covid.cases”.

For the multiple instances APIs, we test by calling for the first twenty results of our main APIs. We test for the total number of instances and the number of returned instances. Then, we ensure that each result has the proper fields and furthermore if these fields were being populated correctly, for example, a string for the “capital” attribute and an integer for the “id” attribute.

The general search tests were similar to the automated tests we ran in Postman, first testing to make sure we got a status ok response from the API. Afterwards, we would test to make sure the results were actually being filtered by checking the count of results and checking the first 10 results to make sure the results were actually being fetched properly.

Postman (backend)

The tests in Postman test whether or not our API calls are functional in a similar manner to our Python backend tests. For each of our single instance APIs, we create a request to the API for a single instance of that model. We test for the presence and type of each of the specific attributes of that model in the response.

For the multiple instances APIs, we perform multiple tests, including the presence and absence of filters, sorting, and search queries. We test for the total number of instances and the number of returned instances. Then, we ensure that each result has the proper fields and types. For checking if filtering is working correctly, we check that the filtered attribute is present in all the returned instances. For sorting, we check that the returned instances match the sorted order. For search queries, we cannot directly check if the API is functioning correctly because the attributes that match the query may not be returned, so we just check that filters and sorting is not being applied to the search queries.

Gitlab CI

For continuous integration with Gitlab CI, currently we use nikolais/python-nodejs or

python:latest docker images to set up the environment for the test jobs. During development, we run all the tests against the localhost version of our website that is booted up within the spawned docker containers for each test job. However, before final submission for the phase, we run the Splinter GUI tests on the deployed version of our website because of imperfect testing with localhost setup in a container. The rest of the tests are still run against the localhost version of our website.

User Stories

Assigned User Stories

Phase 3

Add titles of related instance pages for each instance page, instead of "See Related ."

- This will allow users to see the type of related data there is without having to click a link.

Instead of positive and negative latitude and longitude values, the longitude and latitude should be described with cardinal directions, i.e. North, South, East, and West.

- This will allow users to get a better sense of where the countries are located.

For the user to experience a better UI and focus their view on the center of the page, the pagination buttons should be centered.

- To allow users to understand the buttons functionality more clearly, the buttons should be labeled as First, Prev, Next, and Last.
- To provide more accessibility of the pagination buttons to the users, they should also be included at the bottom of the page.

For filters on number values such as latitude/longitude and ratings, allow filters to be made as ranges of number values. This includes a filtering on whether an attribute is greater than or less than a value, or between two different values.

- This will allow users to interact and filter the data more freely.

While performing a search on model pages or global searches, queries should be able to match not only the names of instances, but also their text attributes as well. For example, when searching for "International", "Haitian Education & Leadership Program" will be a match because of its category in "International Relief," and "KickStart International" will also be matched because of its name.

- This will allow users to search more generally without having to know specific names of instances.

Phase 2

As a user, I would like to see a movable map for the country. The static image does not allow me to explore the area around the country. Adding a map will give more interactivity to the website.

As a user, I would like the data to be more connected so I can find out more news about charities and vice versa.

- More than one related news source in charities instance pages
- More than one related charity in news source instance pages

As a user, I want to be able to paginate through the data easily and accessibly, using the following functionality:

- Next page button labeled "Next"
- Previous page button labeled "Previous
- First page button labeled "First"
- Last page button labeled "Last"
- All buttons included at the top and bottom of the page
- Numbers displaying the current page and the total number of pages

As a user, I want the instance pages and about pages to be more visually appealing.

- The images need to be larger in size so that they can be properly viewed.
- The links need to be clickable for the charity pages and news source pages.
- Can use the space more effectively to fill the entire page.
- Potentially add videos of some form to the charities page

As a user, I want the formatting of the text and data on the model pages to be prettier and better formatted.

- News Sources Model Pages:
    - Categories in news sources should be capitalized
- Charities Model Pages:
    - Rating is changed to Rating (out of 5)
    - Numbers in Income Amount and Asset Amount are comma separated for better readability
- Countries Model Pages:
    - Population value is comma separated
    - Latitude and Longitude are with degree symbols instead of array notation
    - Gini has a link to the wikipedia article describing Gini

<u>Phase 1</u>

As an activist, I want to find more ways I can support my causes.

- The website should show many different kinds of charities with various attributes.
- Scrape data from a data-rich API for charities and provide content on the website about these charities.

As a first-time donator, I want to inform myself of the different types of charities there are and the issues that I might be interested in supporting.

- The website should provide information on different issues through new articles.
- Scrape data from an API for news articles and display this data on the website on the instance pages.

As a citizen of a country, I would like to learn about the news about the country I live in and other foreign countries my country has relationships with.

- The news article pages should provide information about which country the article is about and link to these instance pages for the country.
- Provide links on the news article instance pages to the country instance pages.

As a social media influencer, I would like to be able to direct my followers to charities that I support and have information about them so that I can select a charity that closely aligns with my beliefs.

- The website should provide charity pages that link to their respective article pages so that people can know about the issues these charities support
- Provide links on the charity instance page to related news article instance pages.

As an executive of a large business, I would like to research reputable and well-known charities with widely supported causes for my company to donate to for public good and tax write-offs.

- The website should display information on the charity instance pages that serve as a metric for how reputable and well-known the charities are.
- Provide information on the charity instance pages about how many people they serve and how many donations they receive.

Received User Stories

Phase 3

Allow us to sort columns of the airplane routes so we can see origin and destination countries in alphabetical order. Find some way to sort flight numbers (possibly ascending and descending). Include mechanism to toggle between different sorting methods in the frontend. Would be helpful when we are looking for specific flight routes that we want to keep track of.

- We added sorting functionality for all the columns in the airplane routes table.
- The sort column and direction can be toggled by pressing the name of the column.
- We added sort for departure, arrival cities and countries, and also for flight numbers.

Would like to be able to see only results for a particular city or country. Implement form on front end to allow users to filter by row in the table for countries. Would prefer some sort of checkbox drop down menu to choose filtering choices. Would be helpful when we are looking to see all the results for flights originating from a specific country and other similar queries.

- We implemented the filters as dropdown menus instead of free form inputs because there are a limited number of cities and countries for both departures and arrivals.
- As of now, we have only the option to select a single value from each dropdown so we do not have checkboxes.

There are many different results and we'd like to be able to search for key terms so we can find flights, countries, or articles more efficiently instead of needing to scroll through model page results. Implement a search function so users can enter queries and get results leading to instance pages.

- We added search functionality to all the model pages and also a global search that searches across all model pages.

- The queries are split across whitespace as delimiter to allow for more fuzzy search.

We would like to filter results on the model pages with cards countries and articles). We would like to be able to filter by the attributes displayed on the cards. Also, implement the mechanisms for users to select attributes to filter by with a drop down form. Be able to filter by multiple attributes. Would be helpful when we look at countries and news articles we want to see only a subset because there are so many results.

- Once again, we implemented the filters as dropdown menus instead of free form inputs because there are a limited number of values each attribute can assume.
- We also implemented range filters for numeric attributes in the country page.

Allow for sorting on the model pages. Would like to be able to sort the attributes in ascending or descending order and display the cards based on user choice. Create a mechanism on the frontend to allow for choice on sorting. Sorting would allow us to see countries or articles or flights in a different order which gives us more options for perusing data.

- We added sorting functionality for all the attributes in the countries and articles pages.
- The sort column can be toggled by selecting from the sort dropdown.
- The sort direction can be toggled by toggling the arrow next to the sort dropdown.

<u>Phase 2</u>

We would like to see more detail in the image on the news instance pages. Make the image larger so users can see that image in better detail. Would improve the way the website looks and allow users to get more information about the news article they are clicking on. Potentially make this a different image than the one on the news model page for the card that corresponds to this article.

- We increased the size of the image on the news articles model page.
- We couldn't show a different image as we do not have the data, but we did add an additional image of the news source logo, and also added another media (relevant YouTube video) dynamically.

On the about page, the section to the left that includes the about and other resources appears to be pushed too far to the right. Move over a little for ease of reading for the users and to improve the layout of the page. Just a minor cosmetic change.

- We changed the ratio of the left and right sections on the about page.
- We also added other stylistic changes to the about page to make it easier to read, especially the profile cards.

Set up a database using the method of your choice so you can add more instances to each of your models. Currently there are only 3 instances for each model. We would like more information and want to see more instances for each of the models. Pull this information using your API and database, then modify the pages to programmatically add these new instances to both the model pages and their own instance pages.

- We set up our database on Amazon RDS and also create API routes to access the

database. The API routes also implement the pagination as mentioned in the next story.
- The frontend consumes the APIs exposed by our backend and displays a lot more model pages now than earlier.

Add more instances to each of the model pages. Make sure we can see some results and go to the next page to see more results to make the view easier for a customer looking to see a lot of data. Include the buttons to move between pages at the top of the model page above the table or grid. Should be able to move forward and backwards between pages of results and should also be able to jump to a specific numbered page of results.

- We added a sophisticated pagination system that is available both on top of the model page and the bottom of the model page. It allows skipping to the first or last page, previous or next page, and also selecting a specific page.
- We added selecting the size of the page in addition to the request of the customer.

The color chosen for the favicon is difficult to see. The grey background with the grey icon makes it difficult to tell what the image is. Revise to be a different color for aesthetic purposes.

- The color of the favicon was changed accordingly to a color that agrees better with the color of the browser tab.
- The logo on the navbar was also changed accordingly.

<u>Phase 1</u>

Have a navigation bar at the top of the website that makes it easy to navigate between different pages.

- Make sure the website has a navigation bar at the top of the website that makes it easy to navigate between different pages. Make sure to include a splash page. The navigation bar should lead to your 3 model pages and the about page. Also, customize the splash page and make the navigation bar persistent so users can access pages from anywhere on the website.
- We set up a react-router to link separate pages of the website and create links in the navbar to be able to navigate them.

Have a grid on Model Pages.

- Use a grid or table to show information for your 3 models. Make sure you have 3 instances of each. Make sure the user can navigate to these models.
- We used react-bootstrap Table, CardDeck, and Card components to show model data as grids and tables. On the tables and grids, there are links on each row and card to view each instance page, and there are links in the navbar and splash page to view these model pages.

Have domain name be indicative of project.

- Users should understand the purpose and content provided by the website from the domain name.
- We selected a domain name related to travel and registered it with Namecheap.

Support HTTPS.

- Make sure the website supports HTTPS. When a user types in the domain without a protocol specified make sure it defaults to HTTPS.
- We made configurations in Namecheap to ensure that HTTPS protocol is supported.

Ensure the website is responsive using Bootstrap.

- Use Bootstrap or other tools to create a responsive website.
- We collapsed the navbar in mobile view and made sure content fills up the entire screen without space wasted that could shrink the content.

Hosting Website

AWS

The entire React-based web application is being hosted using Amazon Web Services (AWS), including the frontend UI and the backend server that will be expanded in the future. The following are the components of AWS that are currently being utilized to host all the parts of the web application:

An SSL certificate was obtained from the Certificate Manager to validate the HTTPS website associated with "travelscares.me". This certificate certifies the domain name and also the www web server configuration. The build files of the frontend application are stored in an S3 bucket. Local changes are synchronized with the S3 bucket when a new version of the frontend application is ready to be deployed.

Cloudfront is the Content Delivery Network (CDN) that allows for caching of dynamic content locally for users of the web application. It utilizes the static build files from the S3 bucket to provide a working application hosted on a Cloudfront domain that can also be associated with an external domain name. Certain subtleties with the Cloudfront configuration allow for a smoother user experience include:

- The cache needs to be invalidated every time new build files are available in the S3 bucket. The root of the web application (index.html) is invalidated every time new build files are available. This allows for the latest content to be available to the users.
- The routes for the pages in the web application are statically determined by the react-router. This causes a 403 forbidden error when a page (such as "domain/about") is refreshed or directly accessed as this is not a route handled by the backend. To combat this, a redirection from the 403 error page to the root of the web application is in place, which allows the react-router to appropriately parse the route.

The web application's backend environment configuration is handled by Elastic Beanstalk (EB). The server's location, capacity configuration, and load balancing are part of the environment. The docker image associated with this backend environment is stored in the Elastic Container Registry (ECR) and is synchronized with local changes when a new version of the backend application is ready to be deployed.

The web application's backend environment is hosted on an EC2 instance. No direct involvement with EC2 is required as this environment's configuration is handled internally by Elastic Beanstalk.

The web application's database is hosted in the Relational Database Service (RDS). The database type chosen is MySQL, a purely relational database. The RDS database is accessible publicly and is also connected to the running backend environment.

Namecheap

AWS exposes the frontend application through a Cloudfront domain that is fully functional. However, this is not an easily recognizable name and is usually associated with an external human-readable domain name. The domain name “travelscares.me” was obtained from Namecheap and has been associated with the running web application on AWS. The following routing was required:

- The SSL certificate’s CNAME records for the validation of the domain name and the associated www web server configuration were added to the Namecheap DNS.
- An additional two CNAME records were added to the Namecheap DNS that routed the domain name and the associated www web server configuration to the Cloudfront domain exposed by AWS.

The Namecheap domain and AWS services in conjunction allowed for the creation of a fully functional and SSL-secured web application.