Syrian Refugee Crisis: Phase 3

IDB Group: 18
Shadow Sincross, Matthew Albert, Anish Madgula, Matthew Castilleja

Motivation / Overview

Increasing outreach for Syrian refugees is a pressing humanitarian imperative. It entails providing essential resources like food, shelter, healthcare, and education to those who have been displaced by conflict. Outreach efforts also extend to helping refugees integrate into host communities, offering mental health support, facilitating employment opportunities, and providing legal assistance.

Our project serves to assist Syrian refugees with resources as well as educate the public on the Syrian refugee crisis. We will be providing charities and organizations across many countries which can assist Syrian refugees with housing, medical care, education, etc. This site can be used as a great resource to help increase Syrian refugee outreach across the globe. To build this website, we have utilized JavaScript (React framework), CSS (Bootstrap), HTML, and AWS. We have also utilized public APIs from various sites, including UNHCR, Wikipedia, and Relief Web. In Phase 2, we have added a Python (Flask framework) backend with a MySQL database using SQLAlchemy.

Models

1. Charities
    a. Attributes:
        i. Name of organization
        ii. Description
        iii. Organization Type
        iv. Logo
        v. Image of Organization
        vi. Year established
        vii. Parent organization
        viii. Awards received
        ix. Location of headquarters
        x. Awards Received
        xi. Youtube Information
        xii. Relevant countries
        xiii. Relevant charities
    b. Media:
        i. Logo of charity
        ii. YouTube Video Carousel

2. Countries
    a. Attributes:
        i. Country name
        ii. Capital
        iii. Number of Syrian refugees in country
        iv. Number of Asylum decision in country
        v. Year of Asylum decisions
        vi. Number of individuals granted asylum
        vii. Number of applications rejected for asylum
        viii. Relevant Charities
        ix. Relevant News and Events
    b. Media:
        i. Image of Country's flag
        ii. Interactive Map of country
3. News/Events
    a. Attributes:
        i. Title
        ii. Date of event
        iii. Location of Event
        iv. Disaster type
        v. Media source
        vi. Themes
        vii. Relevant Charities
        viii. Relevant Countries
    b. Media:
        i. Image of event
        ii. Video related to event

User Stories

Phase 1:

We created issues for all of the general requirements that were detailed for this phase. We put them on an issue board and divided them into various labels/categories, including Customer, Frontend, Backend, General, and Project Prep.

Customer Issues:

We received 4 issues from our customer team. They involve collecting a lot of data that is not accessible from our current sources and complex features that require more time than was given for Phase 1. We will continue to work on them across the next few phases while maintaining a dialogue with the Customer team.

Received Stories:

#23: Refugee Assistance Locator - The website homepage has a prominent feature (maybe a search bar or a map) where refugees can input their current location. Upon input, a list or map of nearby charities/organizations is displayed with basic details and contact information. Clicking on a charity/organization opens a detailed view with attributes mentioned (like Year founded, Purpose, Link to website, Media, etc.)

Response: While this could be a really helpful feature for our website, it is currently out of scope for our project given that we do not have some of the attributes that are required to locate the charities and showcase the closest ones based on location. We will try to work towards this feature in future phases of this project.

#24: Education Portal - A dedicated section/page on the website provides a chronological account of events leading to the refugee crisis. Relevant news/events related to each major event are linked appropriately. Visual aids (photos, maps, infographics) accompany the timeline for better understanding.

Response: Creating a timeline of events leading up to the refugee crisis is a great idea that would bring a lot more awareness about this refugee crisis. This website feature/page was completed in Phase 2.

#35: Charity/Organization Spotlight - Feature detailed profiles of different charities and organizations making the largest impact in assisting Syrian refugees. Include content like videos and images showcasing the work and impact of the organization and why they are being placed in the spotlight.

Response: While we think that the idea of a "spotlight" on specific outstanding charities has good merit - we do not believe that it would be easy to rank the efforts made by such in such a quantifiable way.

#36: Refugee Stories Section - Dedicate a section of the website to share personal stories and testimonials from Syrian refugees. Incorporate content, such as video, audio, and even photo essays, to bring the stories to life. Link each story to related charities/organizations and news/events to provide context and avenues for assistance.

Response: While the Refugee stories section is a wonderful idea to implement further on in this project, it is unfortunately out of the current scope. We would need to have more data than we currently have and implement functionalities that would take up an inordinate amount of time.

Phase 2:

At the time of writing this section of the report, we just received 3 customer stories (within 6 hours of the deadline). Due to the amount of time remaining, we were only able to complete 2. We also closed 3 user stories that were given to us in the last phase.

Received Stories:

#59: Add styling for texts on the cards (Bold text, Colored, better font, or whatever fits the site theme best)

Response: We have made some changes to some of our text fields in order to make the cards look more appealing to the users. However, throughout the following phases, we will continue to work on the aesthetics of the website and make it as user-friendly as possible.

#58: Add Model cards that link to model pages on splash page (optional)

Response: We currently do not think there is a need to have another way of navigating to the model pages since we already have implemented the navigation bar at the top of each page. We believe that this is already intuitive and makes the user experience pleasant. **(Closed)**

#57: Add pagination for model pages

Response: We are currently almost done working on this feature, and we will definitely have all the models paginated by the end of this phase. **(Closed)**

Phase 3:

We received the user stories for this phase a couple of hours before the deadline, so we did not have enough time to implement them. We will continue to work on them over the course of the next phase for this project.

Received Stories:

#80: Search bar in navbar

Response: We have created a search bar in our nav bar at the top that allows for site-wide searching. This then redirects the site to our search results page, which has results from all of the model pages. **(Closed)**

#77: Center and style Pagination for News/Events

Response: Due to the limited amount of time left in this phase, we will not have a chance to complete this issue by the deadline. However, we will implement this in the upcoming phase.

#78: Center and style Pagination for Charities

Response: Due to the limited amount of time left in this phase, we will not have a chance to complete this issue by the deadline. However, we will implement this in the upcoming phase.

#79: Center and style Pagination for Countries

Response: Due to the limited amount of time left in this phase, we will not have a chance to complete this issue by the deadline. However, we will implement this in the upcoming phase.

#81: Make model cards the same dimensions

Response: Due to the limited amount of time left in this phase, we will not have a chance to complete this issue by the deadline. However, we will implement this in the upcoming phase.

Restful API

We created our postman api documentation by importing from the example api documentation. All of the models can be accessed through different requests. Each of our models has one request with one example request, as well as their individual instances.

Documentation: https://documenter.getpostman.com/view/30070229/2s9YR9YsK4

Charities API Endpoints

- Retrieve specific charity data by its name
  - GET https://api.syrianrefugeecrisis.me/charities/:charityName
- Retrieve all charities
  - GET https://api.syrianrefugeecrisis.me/charities
    - There are a variety of optional API parameters that can be used for searching/sorting/filtering charity instances
    - These parameters are detailed in the documentation

Countries API Endpoints

- Retrieve specific country data by its name
  - GET https://api.syrianrefugeecrisis.me/countries/:countryName
- Retrieve all countries
  - GET https://api.syrianrefugeecrisis.me/countries
    - There are a variety of optional API parameters that can be used for searching/sorting/filtering country instances
    - These parameters are detailed in the documentation

News and Events API Endpoints

- Retrieve news/event data by its title
  - GET https://api.syrianrefugeecrisis.me/news-and-events/:title
- Retrieve all news/events:
  - GET https://api.syrianrefugeecrisis.me/news-and-events
    - There are a variety of optional API parameters that can be used for searching/sorting/filtering country instances
    - These parameters are detailed in the documentation

Tools

Frontend:

- React: Main UI framework
- React-Bootstrap: CSS Framework
- React Router: Library for implementing website routing/navigation

- NameCheap: Domain name registrar
- AWS Amplify: Cloud-based hosting service
- NPM: Package manager for Node.js
- Unsplash: Provides free images for commercial and personal use
- Jest: Frontend Testing
- Selenium: GUI Testing
- Axios: Used to make HTTPS requests to API
- Google Maps API: Used to render interactive maps for countries
- React-Youtube: Used to embed YouTube videos into charity pages
- React-highlight-words: Used for highlighting query terms in search results
- React-spring: Animation Library for building interactive user interfaces

Backend:

- Python: Language used for backend
- Requests: Python library for HTTP requests
- IPython Notebook: Used for developing Wikidata scraping script
- Flask; Main backend framework
- SQLAlchemy: ORM mapping system between Python and SQL
- MySQL: Relational database used for this application
- Postman - API Documentation/Testing
- AWS RDS; Hosted relational database
- AWS EC2; Hosted API
- Google Geocoding API: Used to collect coordinate information for countries
- YouTube Data API: Used to collect relevant videos for each charity

Hosting

Our project is currently hosted using AWS Amplify. We connected our GitLab repository to our Amplify project to set up continuous deployment of our app. This means that with each commit, the changes were immediately available on our project's domain name. We also obtained our domain name using NameCheap. In order to connect our domain name to our Amplify project, we created a hosted zone through Amazon Route 53. We then configured our name servers on Namecheap and finally added the domain to our Amplify project. Route 53 was able to handle all of the verification of our domain. Amplify also allows us to modify our project's build settings to fit our specific requirements. These settings include commands to be executed at the start of our build, and the directories that they should be executed in.

During phase 2, we also had to connect our backend to AWS, which we did by creating an EC2 instance on AWS and connecting it to our application. We also connected our application to a AWS RDS instance to maintain our MySQL database in the cloud. The RDS service was also connected to an AWS EC2 instance. This allowed us to make sure our application is scalable and has high availability, while providing an easy way to connect to our Flask application.

Details

Frontend Architecture

src Directory:

index.js

- Contains the entry point of the application and logic for rendering all the necessary components that make up the website.
- Defines the overall route structure of the website

src/Components

- Contains the React components that make up the website
- Each page of the website (about, home, charity model, country model, news/events model, instances) has its own component that is called from index.js to render
- Key Components:
    - GenericModelPage.js
        - This component defines the general structure of each model page (i.e. title, search bar, instance grid)
        - This component is called from each specific model page component (CountryModelPage, CharityModelPage, NewsEventsModelPage) with props that include the title of the model, the card component that should be displayed for each instance of the model, and the json formatted data associated with the instances of the model
        - We designed the components associated with model pages in this way in order to allow for more code/component reuse
    - About.js
        - To use the Gitlab API, we used several asynchronous functions to get the user commits, user issues, and total commits/issues
        - Utilized several public Gitlab endpoints (documentation on their website) to gather this information - called these endpoints within the asynchronous functions
        - Utilized useState and useEffect to update the commits/issues info and render the component when a change is made
        - Organized all this information into various Cards using a generic TeamCard.js component and a TeamInfo.js file
    - *ModelPage.js
        - In each of our model pages (NewsEventsModelPage, CountryModelPage, CharityModelPage) we have a callback function named requestInstances that handles formatting API request URLs and using Axios in order to make one request that counts the total number of instances returned for the specific

request, and another request that returns a single page of instances
        - This function is executed when the current page of the model page changes and the useEffect (which contains the requestInstances call) is activated
    - Timeline.js
        - A component that utilizes requestInstances much in the same way that NewsEventsModelPage does
        - Displays a vertical bar that the user can scroll down and see the major news or events corresponding to the Syrian refugee crisis shown in descending order according to the date
        - Has an associated component called TimelineBackground.tsx, which calls Timeline, and displays an interactive background behind it as the user scrolls.
    - SortDropDown.js
        - A component that holds a dropdown for sorting and filtering mechanisms. Uses methods handleSort and HandleFilter respectively that are defined in a model pages .js file, that then passes those functions to the Generic Model page which then passes it to SortDropDown.
        - Utilizes useEffect that sets the lists for sort and the maps for filter based on props. model.
        - Based on the variables held within those two data structures, dropdown items are dynamically created
        - Passes back a string for sort, and two strings for filter (filter category and a filter subcategory)
    - Timeline.tsx
        - The actual component that is called when Timeline is clicked on the navbar
        - Utilizes the animation library “React Spring” to create a Dynamic background
- For Phase 2, we updated the components that required instance data, to retrieve that instance data using a call to our API. We used the Axios JavaScript library in order to make the HTTPS requests to the API.
- For Phase 3, we updated our model pages to have filter and sort dropdown menus to enable filtering/sorting. In addition, we created a search bar in the navigation bar of the website that can be used for sitewide searching and redirects the user to a page that displays the search results of all 3 models.

src/model_data

- Directory contains the json files that stores the scraped data for instances of each model
- This was necessary for Phase 1 as a database has not yet been setup

- For phase 2, we are now accessing our API endpoints to retrieve the information from the database using axios rather than reading from a json file

src/media
- Contains the downloaded images that are used for the carousel (slideshow) on the home/splash page

Backend Architecture

backend (main directory):

application.py:
- Establishes a connection to the RDS hosted SQL database
- Uses SQLAlchemy to interact with the database
- Defines the classes and attributes that represent each of the tables in the database
- Contains all the endpoints for the API with the appropriate routing
- Contains a function to paginate the items in the query
- Contains all the logic for filtering/sorting/searching

db_utils.py
- Populates the database using the scraped data from json files
- Computes connections between models for relevant instances
- Commits information into database so that the endpoints can access it
- Interacts with the database using SQLAlchemy and Flask_SQLAlchemy
- Contains functions to parse JSON data columns into the TEXT type so they can be searched using the MySQL Fulltext search feature
    - Within these functions, the MySQL search indexes are created, which enables the Fulltext search to function

backend/data_scraping Directory:

data_scraping/models_data:
- Contains the json files where the scraped data was stored
- This was required for Phase 1 since a database has not been created
- This was used in Phase 2 in order to store the finalized instance data before programmatically adding this data to the relational database

country_scrape.py:
- Python script used to scrape data for the countries model

- Data scraped using the UNHCR refugee data API and the Wikidata API
- Packages used:
    - requests: Used to make HTTP requests
    - json: Used to convert python dictionary into json file
- Since phase 1 required 3 instances be collected, we hardcoded the country codes to scrape (this will change when a database in installed and we are attempting to collect more instances)
- In phase 2, we removed this hardcoded requirement and used the UNHCR API in order to retrieve a complete list of countries to scrape
- In addition, map information (mainly coordinates) was scraped for each country using the Google Geocoding API

charity_scrape.ipynb:

- Python script and functions used to scrape data for the charities model
- Data scraped using the Wikidata and ReliefWeb API
- Packages used:
    - requests: Used to make HTTP requests
    - json: Used to convert python dictionary into json file
    - copy: Used to make deep copy of each python dictionary of instance data
- Various functions were written in order to transform Wikidata API responses into more useable data
    - For example, for some attributes that the Wikidata API returns (such as the headquarters location of a charity), the response is not plain text, but the Wikidata page ID
    - Therefore, the get_page_title() function accepts as page ID as an argument and returns the title of the page that this ID refers to
- We compiled a list of charities to scrape in the file charities.json, then scraped data for each charity using this script in Phase 2.

news_scrape.py:

- Python script used to scrape data for the new/events model
- Data scraped using the ReliefWeb API and Bing Image/Video API
- Packages used:
    - requests: Used to make HTTP requests
    - json: Used to convert python dictionary into json file
- The ReliefWeb API was scraped in order to get information about each news/events instance, and the Bing API was used in order to collect media for each instance

*_utilities.py:

- These are various files containing utility functions that aided with our data collection
- maps_utilities.py
    - Used to retrieve coordinate information from Geocoding API

- reliefweb_utilities.py
    - Used to collect various data points from the ReliefWeb API
- wikidata_utils.py
    - Used to scrape and format data from the Wikidata API
- youtube_utils.py
    - Used to scrape relevant videos for charities from the YouTube Data API

Phase 2 Features

- **Model Page Pagination**
    - Model pages now showcase the instances across multiple pages using pagination
    - This helps organize the information better and make it more appealing for the users
    - Utilizing a pagination function on the backend in application.py to paginate the query and return the appropriate info to the front-end
- **Flask Backend with SQLAlchemy**
    - Flask framework is used in conjunction with SQLAlchemy to create the API endpoints, connect to AWS RDS, and interact with the relational database
    - Allows for the front-end to call the backend to retrieve information from the database and display it to users
- **Implementation of API**
    - Last phase, we designed our API, but this phase, we have implemented it in our backend
    - We created each of the endpoints in the application.py file and queried the database using SQLAlchemy
- **Interaction with Database**
    - MySQL database hosted using AWS RDS
    - Utilized Flask-SQLAlchemy to connect, populate, and query database

Phase 3 Features

- **Model Page Search**
    - We utilized the model page search bar that we created in Phase 1
    - In order to enable instance searching, we used a variety of state variables and functions that were updated/executed when users enter a query and initiate the search
    - For example, in each of the model pages, searchQuery is a state variable that stores the user query and the handleSort function is a handler that is passed to the SearchBar component and executed whenever a user presses enter or the search button is clicked

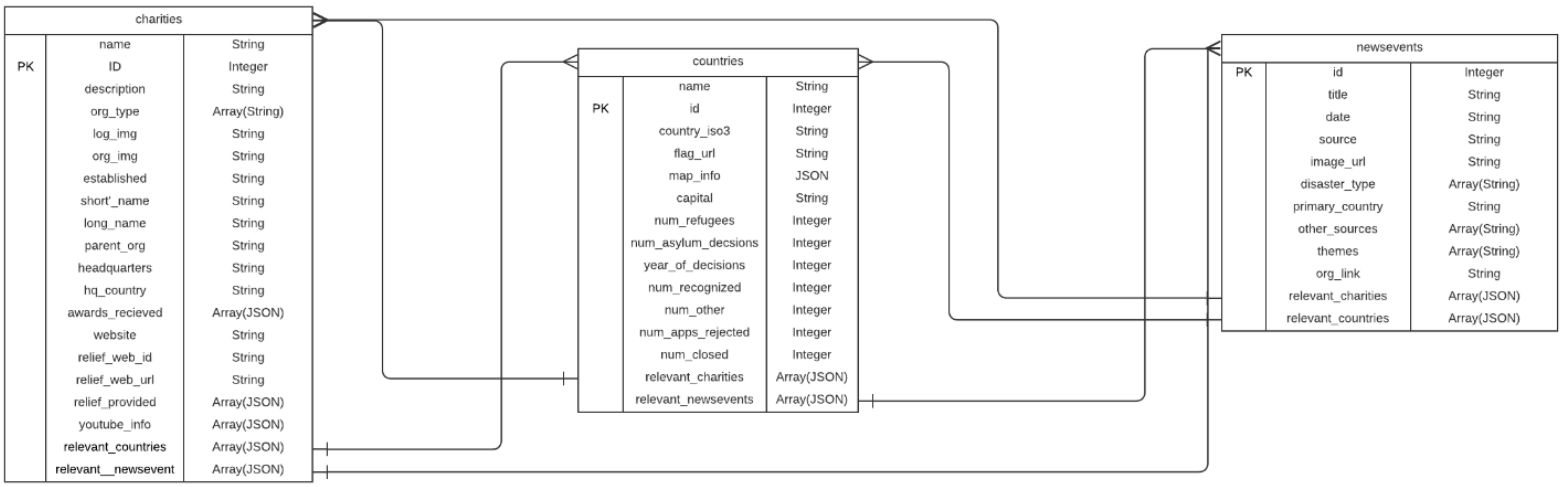
    - We utilized the same pagination functionality from Phase 2 in order to paginate the search results and used react-highlight-words package in order to highlight query terms that appear in search results on the model pages
- **Model Page Sorting/Filtering**
    - Dropdown menu components were created for selecting filter and sort options on the model pages
    - The code associated with these dropdown menus and the logic associated with handling the selection of filter/sort options was done in SortDropDown.js
    - Similar to model page searching, state variables and handler functions (specifically handleSort and handleFilter) are used to store the user-selected sort/filter options and make a request to the API to get the correct instances
- **Sitewide Search**
    - Users can conduct sitewide searches by entering a query in the search bar in the navigation bar
    - Users are then redirected to the search page which is rendered in SearchResultsPage.js
    - The SearchResultsPage component passes the query to the three model page components (which each make a search request to their respective API endpoints) and displays the returned instances for the three models in the same format as the model pages
- **Backend Search Implementation**
    - Utilized MySQL Fulltext search feature in order to allow for google-like searching of instances
    - This Fulltext search feature first required constructing a search index for each model, which specified the columns that would be indexed, and thus searchable
    - In the “get all instances” endpoint of each model, a MATCH…AGAINST SQL query was constructed using SQLAlchemy with the searchQuery argument that was passed in the API request
    - This SQL query was then executed and the results stored
    - In order to enable partial matching in searches, we executed a secondary search if the Fulltext search returned 0 instances
    - This secondary search used the SQL ILIKE operator with SQL wildcards in order allow for partial matching and return instances that contained the search terms as substrings in their data
- **Backend Sorting and Filtering Implementation**
    - Utilized SQLAlchemy filter_by() and order_by() function in order to allow for filtering/sorting of instances
    - Each “get all instances” endpoint utilized a sortBy and sortOrder argument in order to determine which attribute would be used to sort the instances and in which order the sort should be performed in ascending or descending order
    - For filtering, each “get all instances” endpoint utilized several arguments (that were unique to the endpoint) in order to determine which attribute would be used to filter the instances

- Some examples of these filter arguments from the charities endpoint include reliefTypes, hqCountry, and orgType
- Each of these arguments had a JSON list of values that were used for the filtering

- The API endpoints were designed so that if valid sort, search, and filter arguments are present in the API request, then the instances returned from the search will be both sorted and filtered

Database UML Diagram

Our database is structured as shown in the diagram. There are three tables (one for each model) and are connected to each other through the “relevant ___” fields.



Challenges

1. Enabling JSON and numerical data to be searchable on our website and model pages

For sitewide and model-specific instance searching we decided to use the MySQL Fulltext search feature. We chose this feature as it allowed us to efficiently search and rank retrievals based on relevance using a variety of metrics (including search-term frequency and tf-idf weighting). The one problem that we encountered when using this feature is that search indexes can only be constructed from columns that are of the type char, varchar, or text. While a majority of the columns in our model tables were these types, there were several key columns (such as awards_received in the Charities table and disaster_type in the News/Events table) that were a JSON or numerical data type. We wanted these columns to be included in the instance search, but could not include them in the search index with their current data types. To solve this problem, we decided to add columns that were the text data type and programmatically parse the non-text columns into a form that could be represented as text and

searchable. We then added these newly created text columns to the search indexes for each model and were able to successfully search for data contained in these columns.

2. Association Table for representing the relationships between models

Our models aren't as clearly connected as we thought. Determining how closely related instances of different models are is difficult. Regardless, there are many instances of "many to many" relationships in our database. When determining how to connect our instances of different models, we didn't think of creating a completely separate object to represent that relationship. In our next phase, we will modify our database architecture to satisfy the Association Table recommendation. We plan on creating an association table for each pair of our models. We will use the relationship object to store the ids of each connecting instance, as well as key details about their relationship. Additionally, this object can store a score that will represent how strongly connected a relationship is.