CS 373 Software Engineering IDB Group 03

Team Members: Milan Dhaduk, Audrey Tan, Zakaria Sisalem, Pooja Vasanthan

Link to Website: https://wildwareness.net/

Phase 2

WildWareness

- ➢ WildWareness is a web application that provides users with information about wildfires in the state of California. The intended audience is those affected by the fires who need help and/or those interested in helping those affected. Given how recent the issue is, and the severe devastation caused by the fires, we decided that developing this application would be valuable.
- ➢ The website provides information about wildfire incidents, emergency shelters, and community reports (the models). Each has specific instances and information related to those instances. The information is also related in terms of proximity (for example, on a page about a specific wildfire incident, the user can find information about the specific wildfire incident, as well as information on wildfire incidents and community reports that are located near that incident).
- ➢ Users can easily navigate the pages and find the information they seek.

Data Sources and API

- ➢ During the first phase, we decided on using the CAL FIRE , NewsData.io API, Google Places API, Google Maps Geocoding API, and List of Emergency Shelters data sources/APIs to scrape data and get the instances for our model pages. However, during this phase, we realized we couldn't use some sources because of certain limitations.
- ➢ The List of Emergency Shelters didn't provide enough instances or enough information about the instances, so we decided to use Google's Custom Search API and a list of search queries to find shelters in California. We decided to use the Google Maps API and Leaflet APIs to get details about locations and render maps, especially because Leaflet was a free source and was more useful to us than the Google Places API.
- ➢ Moreover, we didn't use the NewsData.io API, because that API only allowed us to fetch relevant news articles from within the last two days of the fetch call, which didn't give us

enough instances. Thus, we found another API called NewsData API that allowed us to get more reports and instances.

- ➢ After our research, we decided to use these APIs/data sources to acquire the following information:
  - CAL FIRE – Reports active and past wildfires with containment details (2025)
  - TheNewsAPI - Information about news/local reports and headlines fetched from TheNewsAPI.
  - Google Maps API- Google's API is used for fetching information about a location.
  - Leaflet- an API used for map rendering given details of a location.
  - Google Maps Geocoding API - a service that allows you to convert addresses into geographic coordinates (latitude & longitude) and vice versa (reverse geocoding).
  - Google Custom Search API- Google's API allows data to be retrieved from custom search results.

Initial Setup

- ➢ Like in the previous Phase, our repo is publicly housed in GitLab. We relied on the issue tracker to track what we needed to implement. In this phase, however, we moved our front-end development to React instead of using HTML to build our website, specifically the frontend folder in our repo.
- ➢ Our development still took place in Visual Studio Code. We added scripts that use various libraries and imports to scrape the APIs and obtain the information we need in JSON format. These JSONs were then used to populate our database.
- ➢ We utilized the Flask framework with SQLite and Flask-CORS to develop our API. SQLite was the relational database engine our database relied on in this phase. However, in the future, we plan to move our database to PostgreSQL.
- ➢ The scripts for data scraping, the database, and JSON data all live in our backend folder in the Git Repo. Other files include our Makefile, the .gitlab-ci.yml file, amplify.yml file, docker-compose.yml file, .gitignore file, README.md, documentation files (Technical Report and UML Diagram), and .env file.
- ➢ Our Makefile contains useful commands, such as pull, push, and status. Some important commands to note are the test commands, which we now use to run the tests we made for our code.
- ➢ The .gitlab-ci.yml file defines the jobs we want to run in our Git pipeline when we push our code to Git. The main additions to this file are the test jobs, which set up the necessary configurations for the respective tests (Jest, Selenium, Postman, etc) with imports and images and then run the tests.

- ➢ The amplify.yml file is relevant to the hosting of our application, and the docker-compose.yml file is used to configure the containers we created in Docker for the frontend and backend.
- ➢ The README.md, Technical Reports, and UML diagram serve as documentation for the work we have been doing towards the development of our website.

Website Architecture

- ➢ The website's basic layout consists of five pages: the Home page, the Wildfire Incidents Page, the Emergency Shelters Page, the Community Reports, and About Page. Each page contains a navigation bar that links to different website pages. Another feature we included is the toggles navigation bar, which appears when the window size is smaller and the navigation bar can't be fully shown at the top.
- ➢ The Home/Splash page contains information about our website: the purpose it fulfills and its intended users. There is an image carousel that the user can navigate through, containing relevant images. In addition, we added cards that link to our model pages, using a similar card framework to the one we used for our instances on the model pages.
- ➢ The About page contains information about our team, such as names, contribution to the project, and education. The Git stats for each team member are acquired using the GitLanb API, and the code for this is in About.jsx. The information about the team members is in card format. Moreover, this page has other information, specifically on the data sources/APIs we used and the tools we relied on throughout our development.

Model and Instance Pages

- ➢ Each model has its page with numerous cards that contain information about specific instances for that model. Each card contains an image representing that instance and at least five sortable attributes of that instance. The card also contains a "Read More" button, which leads to the instance page for that instance. The cards are laid out in a grid format on each model page.
- ➢ Each model page also incorporates pagination. Because there are many instances, having them all on a page wouldn't be convenient for the user. With pagination, we display 10 cards per page, and at the bottom of the page, the user can find how many cards are displayed on the page, the total number of cards there are, the previous and next buttons to navigate between pages, and clickable page numbers to navigate to a specific page.
- ➢ Each instance page contains additional information about the instances in text and multimedia and connections to instances of other models. The connections to instances of other models are based on location (i.e. closest fire and closest shelter to a report). Furthermore, the connections are in card format, like the instance cards. At the end of

each instance page, there's a "Go Back" button leading back to that instance's model page.

- ➢ We ensured all these pages and components are responsive to the window's resizing. We intend to add more information on these instance pages in the future.
- ➢ Model Instances and Attributes:
    - Wildfire Incidents
        - Total Number of Instances: 310
        - Attributes: Name, County, Location, Year, Acres Burned
        - Instance Page Elements: Attributes from above, image, description (text), embedded map of fire location
        - Connections: Nearby shelters and nearby reports.
        - Multimedia: Image, Embedded Map
    - Emergency Shelters
        - Total Number of Instances: 207
        - Attributes: Name, Phone, Address, Website, Rating
        - Instance Page Elements: Attributes from above, image, link to their website, description (text), embedded map of shelter location, reviews (with a view more option to condense the amount of reviews shown at once)
        - Connections: Nearby fires and nearby reports.
        - Multimedia: Image, Embedded Map
    - Community Reports
        - Total Number of Instances: 115
        - Attributes: Title, Source, Date, Author, Categories
        - Instance Page Elements: Attributes from above, image, link to full article, summary/description (text), embedded map of report location, estimated reading time of article
        - Connections: Nearby shelters and nearby fires.
        - Multimedia: Image, Embedded Map

Frontend

- ➢ As mentioned above, we implemented our website in React this time, and all relevant code is in the Frontend folder in our repo.
- ➢ The .jsx files in the src folder contain the code for our website layout. The App.jsx renders the navigation bar and defines the routes to the appropriate model pages and the about page. We used CSS/Boostrap for the styling of the website within the .jsx files, as well as the .css files.
- ➢ The components folder in the src folder holds the main components that were used among the pages: the navigation bar (Navbar.jsx), embedded map (Map.jsx), and

pagination (Pagination.jsx). Because these components were shared among the pages, creating React components made it easier to render them on the pages.
    - Each page on the website contains the navigation bar, each instance page contains an embedded map, and each model page contains pagination. The pagination was described above.
    - We are considering adding a Card component in the future since that's another shared feature among the pages.
- The pages folder contains the .jsx files for all the pages on the website.
    - About Page: About.jsx
    - Home/Splash Page: HomePage.jsx
    - Wildfire Model Page: Wildfires.jsx
    - Wildfire Instance Page: WildfireIncidentsPage.jsx
    - Emergency Shelters Model Page: Shelters.jsx
    - Emergency Shelter Instance Page: ShelterInstancePage.jsx
    - News Report Instance Page: NewsReportInstance.jsx
    - News Reports Model Page: NewsReports.jsx
- With the help of React and CSS, we created a neat and organized layout for our website.

Data Scraping

- The data scraping was done using the Python scripts in the scripts folder in the backend folder. The scripts called the APIs to retrieve information in the form of JSON, which we used to obtain further information. We gathered the retrieved objects in a list and dumped into .JSON files, which are used to populate our database.
- Wildfire Incidents Scraping
    - CAL Fire API, ca_fire_gov.py
    - Process: Obtain the data by calling, then create a dictionary with information extracted from the data, which is added to a list and written to a JSON file.
- Emergency Shelters Scraping
    - Google Custom Search API, shelter_extended.py
    - Process: With a list of California counties, make a search query, call the Google API with the search query, obtain more information on the location using Google Maps API, and get relevant images. Create a dictionary with extracted information from the responses and add to a list, which is written to a JSON file.
- News Report Scraping
    - TheNewsData API, newsdata_api.py
    - Process: Using a long list of search queries relevant to California and the wildfires, call the Google API with the search query based on retrieved

information and perform additional tasks. If there is a URL to the article, scrape that URL heavily to obtain extra information.

- Article URL scraping (All these were done using additional imports and libraries and detecting certain HTML tags)
  - Determine the location of the report by finding mentions of locations in the text of the website (filter out non-Cali locations)
  - Calculates estimated reading time of the article
  - Determine the author of the article (try different techniques and resort to source if can't find one)
  - Find social media links on the page
  - Get coordinates of the locations found using a Geocoder
  - Find relevant articles, images, and videos by performing an additional google search
  - Find the most common words in the text of the article
  - Use an LLM to create a summary of the text
- Put all acquired information in a dictionary and add it to a list that keeps track of the news objects, then write it to a JSON file.
- Some of the information acquired wasn't implemented in this phase, but we aim to implement it in the future.

Backend and Database

- We created our backend and populated our database using the Python scripts api.py, database.py, and models.py files in our backend folder.
- api.py
  - We utilized the Flask framework to build our API. With Flask, we defined routes for our API calls to get information on our instances. The routes are connected to URLs from the website, so when the user "requests" a certain URL it will return the appropriate response using our API.
  - We created two functions (with decorators) for each model: one for getting all the instances of that model and another to get a specific instance of that model using an id. The routes include the names of the tables in our database from which the information is acquired (wildfire_incidents, shelters, news).
  - For the functions that get all instances, a pagination field is included in the response to describe how the pagination will work for that model page.
  - All the routes make use of the GET method
- database.py
  - The database.py file gets the information from the JSON files, filters the information, and adds it to the database by creating a specific instance of it.

    - The information is filtered and converted based on what the database is expecting.
    - There is a function to add an instance of each model type (wildfires, shelters, and reports). The link() function is used to establish the connections between the instances of models based on location in our case.
- models.py
    - This file contains the database layout. There are three tables defined between each of the models (wildfires and shelters, wildfires and news reports, and shelters and news reports) to define the connections.
    - Below these, there is a class for each model specifying the name of the table, the columns of the table (along with the types of those columns (i.e Text)), and an as_instance() function, which returns the information in the form of a dictionary (JSON).
    - For each instance that's created, it's given a unique id called id. This id unique for that instance within its model's table.
- Using all these components, we populated our database

API Documentation

- Our API Documentation stayed mostly the same from Phase 1. We used Postman to design our API. The interface was easy to use and organized our requests clearly.
- All requests defined in the model are GET requests that call our API.
- The main part of the API endpoint: https://api.wildwareness.net/
- We defined two main requests for each model (wildfires, shelters, and news reports). We included one request to get all instances of that model and another request to get a specific instance of that model using id. We included an additional request for retrieving an instance by id. We didn't have any specific query parameters, but the id for a specific instance was defined as an id path variable in our requests.
- Within Postman, we also used the scripts to test calling our API with the requests and checking the object that's returned.
- Link to API Documentation: https://documenter.getpostman.com/view/31322139/2sAYdZvEUy

Hosting

- Following the provided instruction at this repo, we were able to host our flask app on an EC2 instance
- Since the provided url contains instruction for setting up backend hosting, I will only describe the high-level steps taken in order to achieve hosting:
    - Create an EC2 instance, save key in .ssh config
    - SSH into the VM and clone the repo from Gitlab

    - Build the Docker image using the Dockerfile in our ‘backend; directory
    - Run the Docker container which will initialize and start our Flask App (Run detached so will run forever)
- ➢ We implemented HTTPS support afterwards using Caddy as opposed to AWS RDS as described in the provided instructions; Caddy is more lightweight and automatically provides reverse proxy.

Docker Image Creation

- ➢ We created Docker images for our frontend and backend, because we made use of various libraries and imports. The images are based on the imports that our code depends on.
- ➢ We ran the backend Docker container on our EC2 instance which supports our Flask App
- ➢ We implemented a Docker compose in case a developer were to clone our repo and locally test the application

User Stories

- ➢ We provided five user stories to our developer team, who responded well to them all. They implemented the stories or provided just reasoning for implementing an alternative to the story.
- ➢ Our customer team also gave us five user stories. We implemented part of the stories, but some were relevant to later phases, so we let them know that we will implement those in the future.
- ➢ The information about the stories given to us and the stories we provided are in the Git Issue Tracker, along with our conversations/responses to the teams.

Challenges

- ➢ This was a challenging phase, especially since we had to get our frontend running along with our backend, but we managed to surpass most of our challenges.
- ➢ Backend Hosting
    - Backend hosting took a long time to understand and get set up. We used SQLite, so we had issues migrating to Postgre. For this phase, we are sticking with SQLite since our web application still works, but we plan to move to Postgre in the future.
- ➢ Database Connections
    - Making the connections between instances of different models was new. Understanding the format and how we should define the tables took some effort, so the connections between related instances were established.
- ➢ API Scraping

    - API scraping was also difficult. This was because not all the information was contained in the response provided by the APIs we used. We had to write additional code to query for more information and use other APIs, specifically Google's Custom Search.
    - This was especially hard with the news report because most information acquired didn't include sortable attributes that made sense. Thus, we had to spend time scraping the article URL. We had to use various techniques, and sometimes, accessing the URL wasn't successful due to errors such as authorization issues.
- Generating the Data
    - Generating the JSON files with all the information on the instances took a long time. The news report data generation took a long time because the articles undergo heavy processing in the scripts.
    - Additionally, the NewsData API had a limit, so we couldn't generate as many instances as we would've liked, but we have enough.

Toolchains

- React Bootstrap Framework - used to design the website
- Selenium, Postman, Jest, and UnitTest Frameworks for testing
- Python
- Docker
- Google Maps- to obtain locations for certain instances
- AWS Hosting-used to host website and backend
- Amazon Route 53- to obtain URL
- Postman- for API documentation/design
- Google Documents- for technical report documentation
- Visual Studio Code- IDE used for frontend development of the website
- GitLab- houses project repository and make it available for public view
- Grammarly
- Google (Stack Overflow, GeeksforGeeks, Research, etc)
- Slack for Communication

Other Sources:

- HomelessAid- Referenced previous project to develop layout of website.
- ChatGPT- Used to develop the code for the website by asking about implementing features (i.e. embedded maps and videos) in React, how to write python scripts to use the public APIs, how to implement the backend using Flask (explaining what Flask was, etc), and hosting in AWS.