CS 373 Software Engineering IDB Group 03

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

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

Phase 3

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 using filtering, sorting, and searching.

Models and Data

➢ Our application is centered around three primary models: wildfires, emergency shelters, and community reports. Each model contains a distinct set of attributes. Each instance also contains an associated image, map, and additional metadata presented on its detail page.

- ○ Wildfires include data points such as the name of the fire, the county in which it occurred, its coordinates, the year, and acres burned.
- ○ Emergency shelters are characterized by name, address, phone number, rating, and website.
- ○ Community reports store information including title, source, publication date, author, and topic categories.

➢ The SQLite database schema includes individual tables for each model along with three intermediary tables to represent relationships between the models—allowing us to efficiently link shelters to fires, fires to reports, and so on. These connections are made based on geolocation and are established during the data ingestion process. We designed our database using SQLAlchemy ORM, and each class includes methods to serialize instances into JSON for API responses.

➢ In total, our database currently includes 310 wildfire instances, 207 shelters, and 115 community reports.

Website Architecture

➢ The architecture of WildWareness follows a modular, full-stack design. The frontend is implemented using React and styled with Bootstrap and custom CSS. It communicates with a RESTful backend built using Flask, which interfaces with a SQLite database that stores all the relevant model data. Each of these components is containerized using Docker and deployed to AWS EC2. The backend is hosted with HTTPS support via Caddy, a lightweight web server that handles reverse proxying and TLS configuration. The frontend is separately hosted using AWS Amplify, and the application is accessible through a custom domain via Amazon Route 53.

➢ The site itself is divided into several main pages: a home page, model pages for wildfires, shelters, and reports, individual instance pages for each data model, and an about page. Each page is styled to be responsive and accessible across devices. Instance pages are dynamically generated using the ID of the item, with data fetched via the backend API. These instance views also include embedded maps and cross-model links based on geospatial proximity, creating a cohesive user experience that connects different types of information meaningfully.

➢ Component Breakdown:
  - Frontend: React, Bootstrap, CSS
  - Backend: Flask REST API
  - Database: SQLite (for now)
  - Hosting: Dockerized containers on AWS EC2

➢ Pages:
  - Home: Overview, navigation to key models.
  - Model Pages: Wildfires, Shelters, Community Reports.
  - Instance Pages: Detail view with multimedia, map, and cross-model proximity data.
  - About: Team info, contributions, data sources.

Challenges and Solutions

- ➢ Several challenges emerged during Phase II and III. Hosting the backend proved particularly difficult, especially due to the differences between local SQLite setups and production hosting environments. We originally considered PostgreSQL but opted to remain with SQLite during this phase for consistency. Setting up secure HTTPS with Docker also required learning new tooling, which we resolved by switching from AWS RDS to Caddy for its simplicity.
- ➢ Data scraping presented another major hurdle. Many APIs had data limitations or rate caps, requiring us to supplement them with additional sources or web scraping techniques. The community news scraping was especially complex, involving HTML parsing, metadata extraction, and the use of language models to process text content. This required a significant time investment and debugging effort, but ultimately resulted in rich, high-quality report data.

Implementation of Phase 2 & 3 Features

- ➢ During Phase II and III, we significantly expanded our feature set by adding pagination, sorting, searching, and more robust backend-to-frontend communication. All model pages now support dynamic pagination, displaying ten instances per page, with controls to navigate between pages or jump to a specific one.
- ➢ Additionally, we implemented client-side sorting features for key attributes such as fire year, shelter rating, or news date, providing users with a clearer and more customizable browsing experience.
- ➢ We also added a search bar integrated into the navigation bar, allowing users to search across models using free-form queries. This feature filters results based on keywords found in instance titles, descriptions, and other fields. On the backend, pagination logic is included in our API responses, and the frontend fetches only the relevant data for the current page. We maintained route consistency using React Router, ensuring that pagination and sorting parameters are reflected in the URL for a shareable experience.
- ➢ To support these features, the backend API was expanded to include pagination logic in its responses, while the frontend uses dynamic state management to render only the relevant instances based on user input.

Data Collection Scraping

- ➢ The data used in our application was gathered through custom-built Python scripts that utilize various public APIs and scraping techniques. For wildfire incidents, we relied on the CAL FIRE API, which returns up-to-date information about fire locations and statuses. The data was parsed into JSON and enriched with geolocation data using the Google Maps Geocoding API.

- ➢ For emergency shelters, we initially attempted to use official datasets but found them lacking in coverage. As a solution, we utilized Google's Custom Search API in combination with queries based on California counties. We then augmented the results with address, contact, and image data using the Google Maps API.
- ➢ Community reports were scraped using TheNewsAPI. To enrich these articles, our scripts visited the actual URLs and extracted metadata such as the article's author, estimated reading time, location mentions, embedded social media links, and images. When available, these scripts also used an LLM to generate a concise summary of the content. This information was saved into JSON files that were then used to populate our database.

Frontend Implementation

- ➢ The frontend was developed using React and structured into a clear hierarchy of components and pages. The main entry point (App.jsx) defines the navigation bar and routing for different pages. Shared components such as the map view (Map.jsx) and pagination controls (Pagination.jsx) are used across all model pages. Each model and instance has its own dedicated page component located in the pages/ directory.
- ➢ Styling is handled via Bootstrap and modular CSS, ensuring that the site is responsive and accessible on both desktop and mobile devices. For sorting and search functionality, we use React state and props to update the view in real time. The navigation bar was enhanced with a search input that dynamically filters content across models based on user input.

Backend and API Design

- ➢ Our backend is built using the Flask framework, structured around RESTful principles. For each model, we defined two main API routes: one for retrieving all instances with pagination (/model) and one for fetching a single instance by ID (/model/<id>). These routes return JSON objects, and all requests are served using the GET method. Flask-CORS is used to allow secure cross-origin communication between the frontend and backend.
- ➢ The backend includes a dedicated models.py file where each data model is defined using SQLAlchemy. Each model class specifies its columns, data types, and relationship tables. A function called as_instance() is included in each model to convert data into API-consumable dictionaries. The database is populated through scripts defined in database.py, which parse and filter the JSON data before committing it to the database.
- ➢ The backend is containerized using Docker, with images created for both local development and production deployment. For teams taking over this project, a

docker-compose.yml file is included to easily spin up a local development stack with both frontend and backend.

Hosting and Deployment

- ➢ Our backend is deployed using Docker and hosted on an AWS EC2 instance. We SSH into the virtual machine, clone the Git repository, and build the Docker image using the provided Dockerfile. Once the container is running, Caddy handles HTTPS and reverse proxy functionality, providing secure and continuous deployment. For the frontend, we used AWS Amplify, which offers seamless deployment and integration with our GitLab repository. A custom domain was configured via Amazon Route 53.
- ➢ This architecture allows for isolated development and deployment of the frontend and backend, improving flexibility and scalability. Our Docker Compose file is available for local testing, making it easy for developers to replicate the environment on their machines.

API Documentation

- ➢ https://documenter.getpostman.com/view/31322139/2sAYdZvEUy
- ➢ All of our API endpoints are documented using Postman, a powerful tool for building and testing APIs. Each model includes endpoints to fetch all instances and to retrieve individual instances by ID. Pagination metadata is included in the response for GET /model routes. Our API does not currently accept query parameters beyond pagination, but future iterations may expand this. We also wrote scripts in Postman to test the accuracy and reliability of our API responses during development. Full documentation can be found here.

User Stories

- ➢ We collaborated closely with our development and customer teams to define and implement a set of user stories. Developer stories focused on frontend functionality like pagination, sorting, and instance connections. These were implemented successfully and provided an improved user experience. Customer stories emphasized the need for detailed shelter data, proximity-based connections, and readable report summaries. Some features, such as filtering by rating or reading time, are scheduled for future phases.
- ➢ All user stories and their progress were tracked via GitLab's issue tracker. Each story was addressed with comments detailing design decisions, alternatives, and implementation plans.

User Stories

➢ To build WildWareness, we employed a full stack of modern web development tools. The frontend was built using React, Bootstrap, and CSS, and the backend relied on Flask, SQLAlchemy, and SQLite. Data scraping was implemented in Python, with use of TheNewsAPI, CAL FIRE API, Google Custom Search, and Google Maps API. We containerized our applications using Docker, documented our API in Postman, and hosted the project on AWS EC2 and AWS Amplify. Additional tools included Selenium, Jest, GitLab CI/CD, and Slack for team coordination.