United States Natural Disaster Service

Design Document

Table of Contents

[Problem Statement 2](#_Toc192242486)

[Use Cases 2](#_Toc192242487)

[Disaster Details 2](#_Toc192242488)

[Topological Map 2](#_Toc192242489)

[Create Disaster 2](#_Toc192242490)

[Update Disaster 2](#_Toc192242491)

[Project Scope 2](#_Toc192242492)

[In Scope 2](#_Toc192242493)

[Out of Scope 3](#_Toc192242494)

[Open Questions 3](#_Toc192242495)

[UX Design 4](#_Toc192242496)

[Proposed Architecture 4](#_Toc192242497)

[Disaster Frontend Service 4](#_Toc192242498)

[Proposed Solution 5](#_Toc192242499)

[Disaster Backend Service 5](#_Toc192242500)

[Proposed Solution 5](#_Toc192242501)

[Disasters Database 5](#_Toc192242502)

[Proposed Solution 6](#_Toc192242503)

[Technologies 6](#_Toc192242504)

[Authorization 6](#_Toc192242505)

[Frontend Framework 6](#_Toc192242506)

[Map Rendering 6](#_Toc192242507)

[Backend Framework 6](#_Toc192242508)

[Service Design 7](#_Toc192242509)

[Disaster Backend Service 7](#_Toc192242510)

[Internal Models 7](#_Toc192242511)

[Components 7](#_Toc192242512)

[API Design 7](#_Toc192242513)

[Data Design 11](#_Toc192242514)

[Disasters Table 11](#_Toc192242515)

[Tasks and Milestones 11](#_Toc192242516)

[Timeline 12](#_Toc192242517)

[Milestones 12](#_Toc192242518)

[Milestone 1 12](#_Toc192242519)

[Appendices 12](#_Toc192242520)

[A: Definitions 12](#_Toc192242521)

[B: Citations 12](#_Toc192242522)

[C: Alternative Technologies 13](#_Toc192242523)

[Frontend Service 13](#_Toc192242524)

[Backend Service 13](#_Toc192242525)

[Database 14](#_Toc192242526)

# Problem Statement

The United States continues to face a number of natural disasters on a regular basis. The goal of our application is to track ongoing natural disasters across the United States. In the long term, we’d like to expand this service to also provide historical logs and analytical views of disasters.

In this design document, we outline a proposed design for the United States Natural Disaster Service web application.

# Use Cases

## Disaster Details

U1

As a United States Natural Disaster Service customer, I want to get a detailed explanation of a disaster when I open a disaster details page.

## Topological Map

U2

As a United States Natural Disaster Service customer, I want to access a topological map showing ongoing natural disasters.

## Create Disaster

U3

As a United States Natural Disaster Service administrator, I want to create new logs of natural disasters.

## Update Disaster

U4

As a United States Natural Disaster Service administrator, I want to update logs of existing natural disasters.

# Project Scope

## In Scope

* Detailed explanation of a disaster on a disaster details page, including: Disaster ID, Disaster Type Area, Death Toll, Injuries, Financial Impact, Start Date/Time, End Date/Time, Disaster Severity
* Logging the following disaster types: Earth quakes, Tornadoes, Hurricanes, Fires, Floods, Volcanic Eruptions, Blizzards, Tsunamis
* Topological map showing ongoing natural disasters. Clicking on a disaster will show the following: Type of disaster, Start Date, Death toll, Injuries, Severity
* Administrators are able to create new logs of natural disasters
* Administrators are able to update existing natural disaster logs

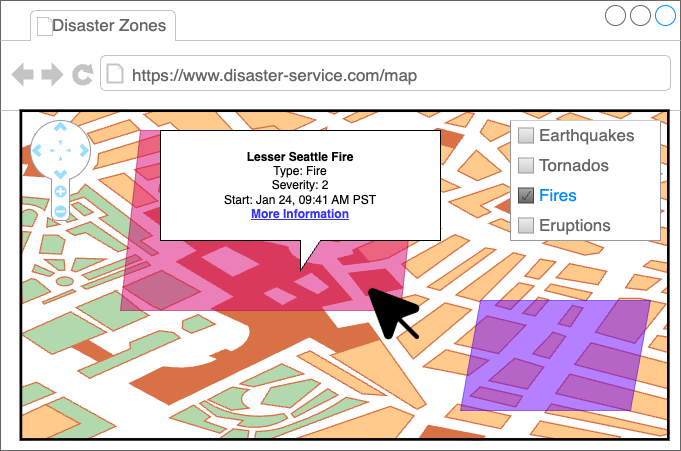
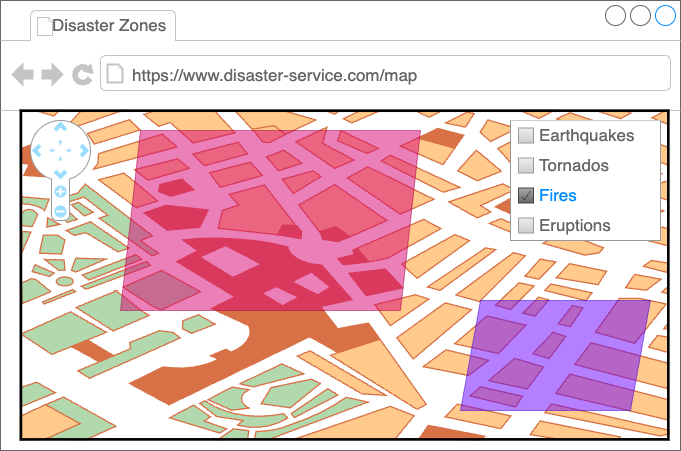
## Out of Scope

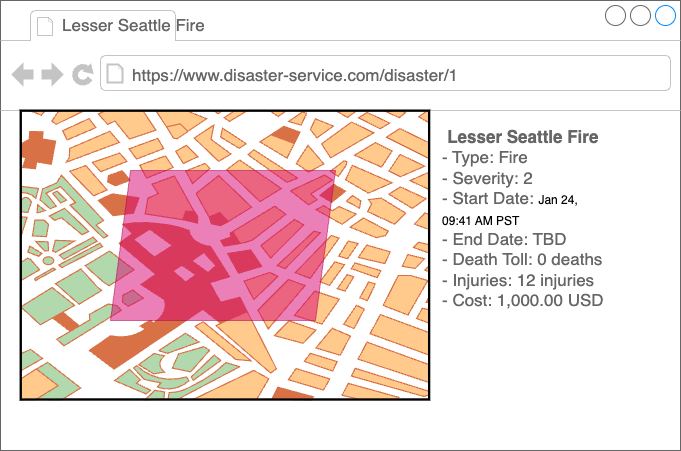
* Disaster history
  + Historical Records
  + Analytical views (Heatmaps)
* Providing disaster services resources
* Non-administrator user tracking
* Integration with natural disaster resources, such as the USGS and NWS
* Disaster forecasts
* Deleting natural disaster events
* Reporting erroneous natural disaster logs
* Map customization

# Open Questions

1. Naming: Should we consider renaming “Disaster” items to “Events”?
2. Do we want to add names to the disaster entries?

# UX Design





# Proposed Architecture

Disaster Service Architectural Diagram

Disaster Frontend Service: Amplify Application

Disaster Backend API: API Gateway Endpoint

Disaster Backend Service: AWS Lambda Function

Disaster Database: RDS using Amazon Aurora Serverless

## Disaster Frontend Service

The Disaster Frontend Service acts as the presentation layer in our three-tier architecture. The Disaster Frontend Service will render the user experience for web and mobile platforms, including rendering maps of disasters, disaster details, and authenticated access to forms for creating and updating disasters.

### Proposed Solution

AWS Amplify

* Pros
  + Fully managed CI/CD
  + Integration built-in for many AWS services
    - Cognito for authentication/authorization
    - Calling other resources using RESTful APIs (API Gateway)
    - Includes Route 53 for domain name resolution
    - Distributed using Cloudfront for CDN
    - Integrates w/ Amazon Location Service
  + Integrations w/ MapLibreGL-JS for Geospatial (Maps)
  + Fully managed scaling
  + Multiple source control / version control options (GitHub, GitLab, BitBucket)
* Cons
  + Working with mostly Typescript / Javascript
  + Opaque (how it works under the hood)
  + Locks us in to Amplify (not very easy to migrate away from)
  + Locked in to certain code sources (GitHub, GitLab, Bitbucket)

## Disaster Backend Service

The Disaster Backend Service acts as the logic layer in our three-tier architecture. This service handles requests for disaster data, performs queries on the disaster database to create, retrieve, update, delete, or list disaster data, and returns the data in the [GeoJSON format](https://datatracker.ietf.org/doc/html/rfc7946), to be used by the frontend. We will use a RESTful API for our Backend Service

### Proposed Solution

Single AWS Lambda Function & API Gateway

* Requests are stateless
* Requests will be handled within seconds
* Fully managed scaling
* 1 million free requests / mo.
* Only pay while invoked
* 10GB memory limit is a minor concern if we have many disasters – we may need to implement pagination / tiling in the future
* Easily integrates w/ API Gateway (including proxy)
* Minimal cold starts

## Disasters Database

The Disasters Database is the data layer of our three-tier architecture. The Disaster Database stores information related to Disasters. Disaster data includes properties such as disaster type, severity, start and end dates for each disaster. Disaster data also includes geospatial data, representing the area of impact for each disaster.

### Proposed Solution

Aurora PostgreSQL Serverless with PostGIS extension

* PostGIS extension for geospatial data
* Single-DB solution (normalization, transactional)
* Serverless – fully managed (no need to worry about increasing storage or compute capacity)

# Technologies

## Authorization

## Frontend Framework

## Map Rendering

## Backend Framework

# Service Design

## Disaster Backend Service

### Internal Models

Class diagram

Disaster Data Class:
property properties: Disaster Data Properties
property area: Disaster Area Data

Disaster Data Properties Class:
property disaster id: Universally Unique Identifier
property deaths: integer
property injuries: integer
property financial impact: floating number
property start date : datetime
property end date: optional datetime
property disaster type: Disaster Type Data

Disaster Type Data Enum:
values:
Fire
Tornado
Earthquake
Hurricane
Flood
Eruption
Blizzard
Tsunami

Disaster Severity Data Enum:
values:
Low: 1
Medium: 2
High: 3

Disaster Area Data Class:
property Coordinates: list of points

Point class:
property lattitude: floating point number
property longitude: floating point number

### Components

Components Class Diagram:

Handler class:
method handle: accepts request, returns response

Activities:

Update Disaster Activity Class:
property disaster DAO
method: create_disaster
Accepts UpdateDisasterRequest
 Returns CreateUpdateDisasterResponse

List Disasters Activity Class:
property Disaster DAO
method: list_disasters
Accepts ListDisastersRequest
Returns ListDisastersResponse

Get Disaster Activity Class:
disaster DAO
method: get disasters
Accepts GetDisasterFeatureRequest
Returns GetDisasterFeatureResponse
 
Create Disaster Activity Class:
property Disaster DAO
method create disaster
accepts CreateDisasterRequest
returns CreateDisasterResponse

Disaster DAO class:
method create create disaster
accepts disaster data
returns disaster data

method get disaster:
accepts disaster id
returns disaster data

method list disasters:
accepts list of disaster types and current filter flat
returns list of disaster data

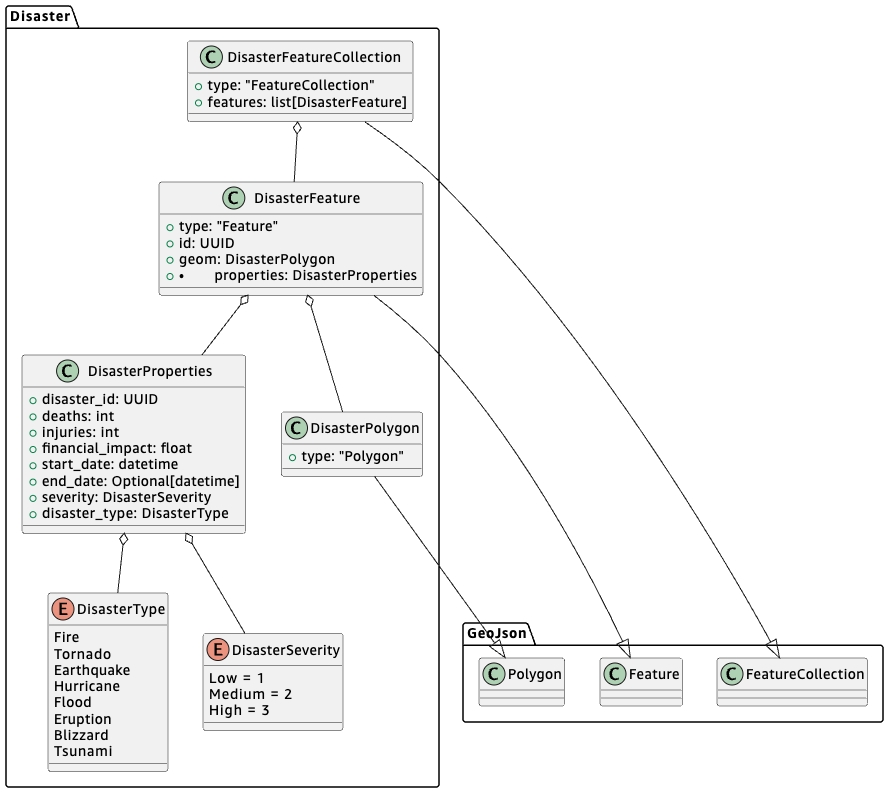
method update disaster:
accepts disaster data
returns disaster data  

### API Design

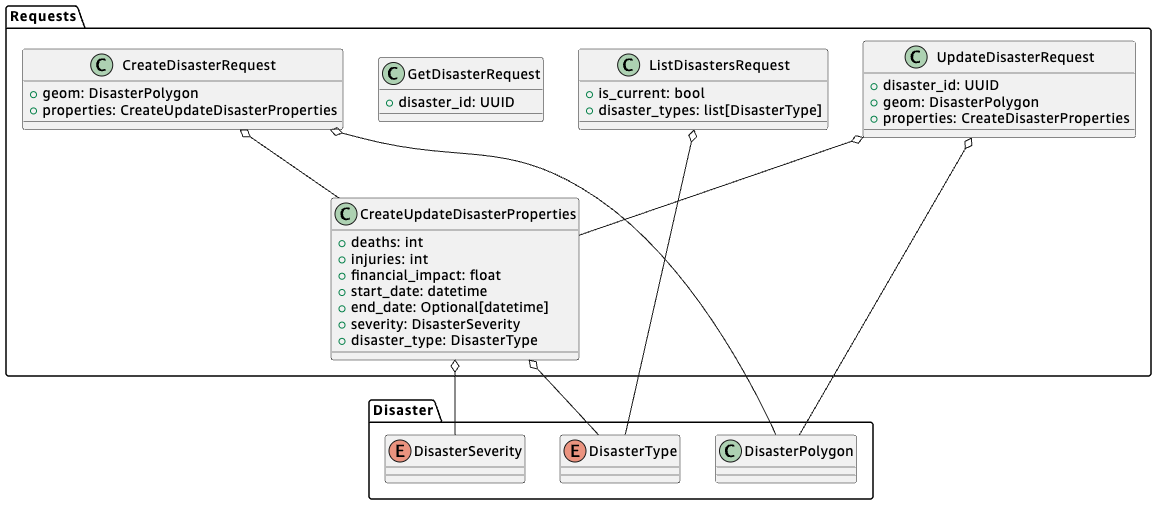
#### Public Models

##### Disaster

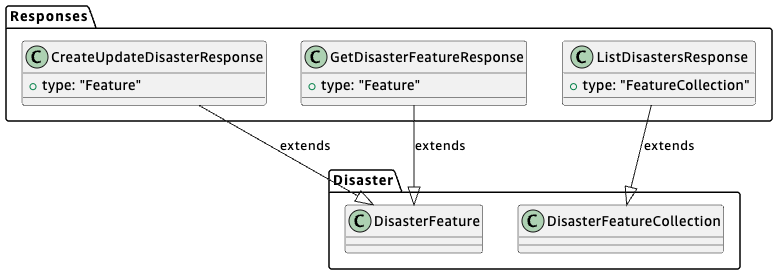
Classes in the “GeoJson” package follow the [GeoJSON specification](https://datatracker.ietf.org/doc/html/rfc7946).



##### Requests



##### Responses



#### Endpoints

##### ListDisasters

Method:

* GET

Path:

* /disaster?is\_current={is\_current}&disaster\_type={disaster\_type}&disaster\_type={disaster\_type}

Request:

* ListDisastersRequest
  + {is\_current} -> is\_current
  + {disaster\_type} -> disaster\_types
    - Multiple: Collect into a list

Response:

* Status: 200
* ListDisastersResponse
  + `features` is empty if no disasters matching query params is found

Error Responses:

* 500: internal server error

##### GetDisaster

Method:

* GET

Path:

* /disaster/{disaster\_id}

Request:

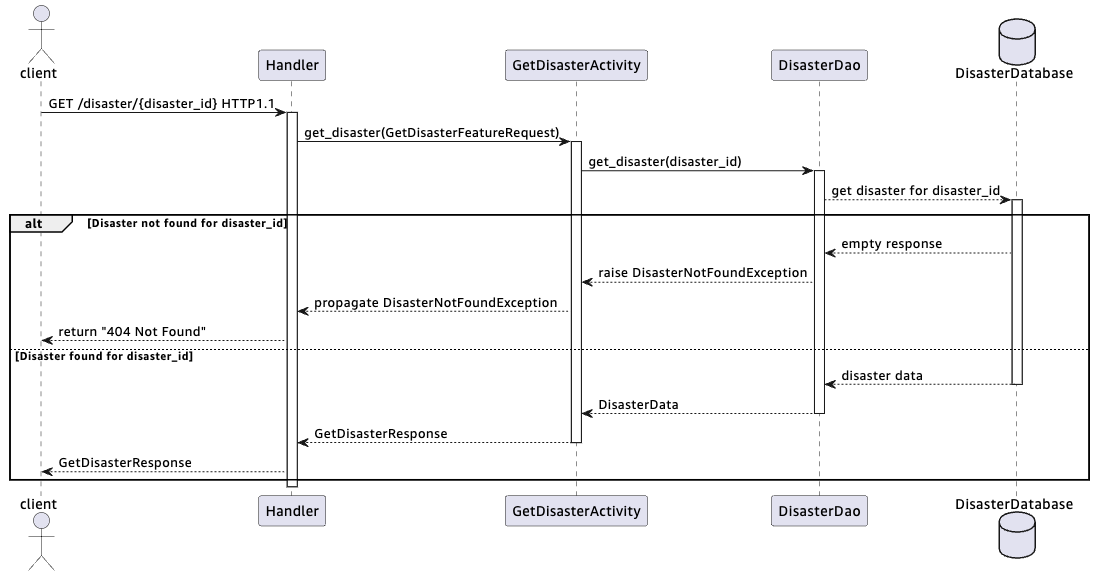
* GetDisasterRequest
  + {disaster\_id} -> disaster\_id

Response:

* Status: 200
* GetDisasterFeatureResponse

Error Responses:

* 404: no disaster found for given ID
* 500: internal server error



##### CreateDisaster

Method:

* POST

Path:

* /disaster

Headers:

* Authorization

Request:

* CreateDisasterRequest (body)

Response:

* Status: 200
* CreateUpdateDisasterResponse

Error Responses:

* 401: unauthorized request
* 500: internal server error

##### UpdateDisaster

Method:

* PUT

Path:

* /disaster/{disaster\_id}

Headers:

* Authorization

Path parameters:

* disaster\_id: str (UUID)

Body:

* UpdateDisasterRequest
  + {disaster\_id} -> disaster\_id
  + Body -> geom, properties

Response:

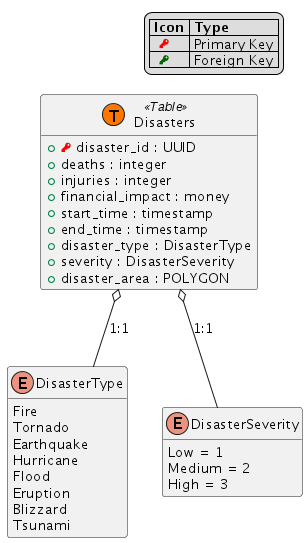
* Status: 200
* CreateUpdateDisasterResponse

Error Responses:

* 401: unauthorized request
* 404: no disaster found for given ID
* 500: internal server error

# Data Design

## Disasters Table



# Tasks and Milestones

*Break down the effort this project will require into tasks and subtasks. Then group tasks into milestones.*

*Tasks should cover a vertical slice of your effort – for example, each API endpoint may constitute a task. Creating the database table may be a subtask, as would creating the API, creating the business logic, and so forth.*

*You may also choose to split some tasks or milestones by horizontal slices – maybe you want to create a separate milestone for creating the initial architecture, or one for implementing your UX.*

*If you need to perform any additional investigation, create tasks for those as well.*

## Timeline

## Milestones

### Milestone 1

Title

#### Task 1.1

Title

##### Subtask 1.1.1

Title

# Appendices

## A: Definitions

|  |  |
| --- | --- |
| Term | Definition |
| Natural Disaster | Earth quakes, Tornadoes, Hurricanes, Fires, Floods, Volcano Eruptions, Blizzards, Tsunami |
| NWS | Natural Weather Service |
| SLA | “Service Level Agreement” – agreement on what will be delivered, and under what timelines. |
| URI | “Uniform Resource Identifier – a system for identifying resources. In our case, this means the path to the resource. |
| URL | “Uniform Resource Location” – a system for identifying locations of resources. Often includes a protocol, a domain name, and a path to the resource. |
| USD | United States Dollars |
| USGS | United States Geological Survey |
| UX | “User Experience” |

## B: Citations

1. H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub. 2016. RFC 7946: The GeoJSON Format. RFC Editor, USA, datatracker.ietf.org/doc/html/rfc7946

## C: Alternative Technologies

### Frontend Service

#### Django

* Pros
  + Python
  + Templating for sites
  + Integrate with JS
  + Libraries available for working with Geospatial data (working with Leaflet library)
  + Libraries for authentication / authorization
* Cons
  + Heavyweight solution (adds datastore support)
  + Would need to extend to support accessing RESTful APIs (backend)
  + Would need to add CI/CD
  + Need to set up DNS w/ Route 53
  + Would need to set up CloudFront for CDN

#### S3 hosting

* Pros
  + Easy to set up bucket for web hosting
* Cons
  + More manual work with HTML/JS
  + Permissions management
  + More work to integration with authentication/authorization
  + Would need to add CI/CD
  + Need to set up DNS w/ Route 53
  + Would need to set up CloudFront for CDN

### Backend Service

#### Multiple AWS Lambda Functions

* Pros:
  + Same as single AWS Lambda Function
* Cons:
  + Code duplication
  + Segmented monitoring/logging
  + More likely to encounter cold starts

#### ECS Fargate

* Pros:
  + Fully managed / serverless
* Cons:
  + More costly compared to AWS Lambda for 1st million requests / mo.

### Database

#### DynamoDB

* Pros:
  + Only need simple queries
  + Schema-less: allows customization of different disaster types
* Cons:
  + Need to perform geospatial analysis manually (stretch goal)
  + Unknown number of points for polygon – can create very large attributes
  + Strong consistency costs more

#### Multiple Databases

* Use DynamoDB for disaster properties, Aurora for spatial data
* Pros:
  + Benefits of both types of databases
* Cons:
  + Ensuring consistency between the two tables is complicated (especially for transactions)
  + Eventual consistency (or pay more) for properties

#### Redis Caching

* Pros:
  + Can add geospatial queries to underlying DB
* Cons:
  + Eventual consistency
  + May need to handle large amounts of data in-memory
  + Adds additional complexity to design
  + Additional costs for caching