United States Natural Disaster Service

Design Document

Table of Contents

[Problem Statement 2](#_Toc193450317)

[Use Cases 2](#_Toc193450318)

[Disaster Details 2](#_Toc193450319)

[Topological Map 2](#_Toc193450320)

[Create Disaster 2](#_Toc193450321)

[Update Disaster 2](#_Toc193450322)

[Project Scope 3](#_Toc193450323)

[In Scope 3](#_Toc193450324)

[Out of Scope 3](#_Toc193450325)

[Open Questions 3](#_Toc193450326)

[UX Design 4](#_Toc193450327)

[Proposed Architecture 4](#_Toc193450328)

[Disaster Frontend Service 4](#_Toc193450329)

[Proposed Solution 4](#_Toc193450330)

[Disaster Backend Service 5](#_Toc193450331)

[Proposed Solution 5](#_Toc193450332)

[Disasters Database 6](#_Toc193450333)

[Proposed Solution 6](#_Toc193450334)

[Technologies 6](#_Toc193450335)

[Authentication 6](#_Toc193450336)

[Frontend Framework 7](#_Toc193450337)

[Map Rendering 7](#_Toc193450338)

[Backend Framework 7](#_Toc193450339)

[Backend Service Design 7](#_Toc193450340)

[Model & Components 8](#_Toc193450341)

[API Design 8](#_Toc193450342)

[Public Models 8](#_Toc193450343)

[Endpoints 9](#_Toc193450344)

[Data Design 13](#_Toc193450345)

[Tasks and Milestones 14](#_Toc193450346)

[Timeline 14](#_Toc193450347)

[Milestones 14](#_Toc193450348)

[Design 14](#_Toc193450349)

[Infrastructure Setup 15](#_Toc193450350)

[Common Components 16](#_Toc193450351)

[Disaster Details Page 19](#_Toc193450352)

[Disasters Map Page 19](#_Toc193450353)

[Create Disaster 20](#_Toc193450354)

[Update Disaster 21](#_Toc193450355)

[Appendices 22](#_Toc193450356)

[Internal Documentation 22](#_Toc193450357)

[Definitions 22](#_Toc193450358)

[Alternative Technologies 23](#_Toc193450359)

[Service Design 23](#_Toc193450360)

[Database 23](#_Toc193450361)

[Map Rendering 24](#_Toc193450362)

[External Documentation 24](#_Toc193450363)

# 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

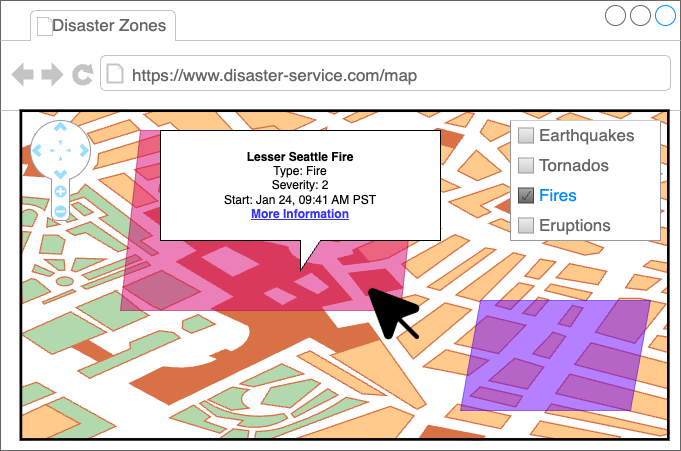
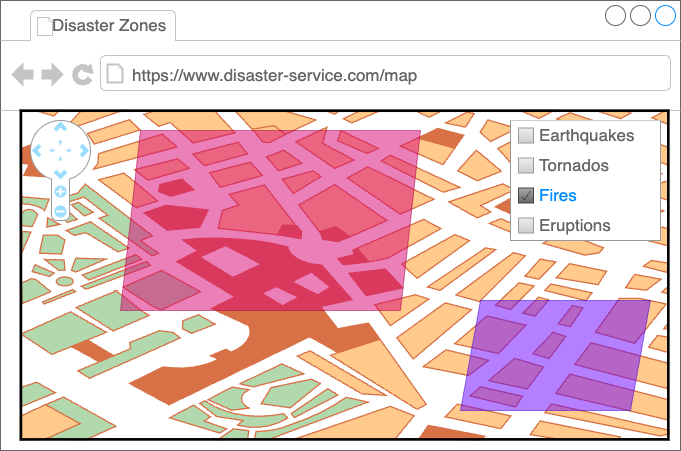
## Out of Scope

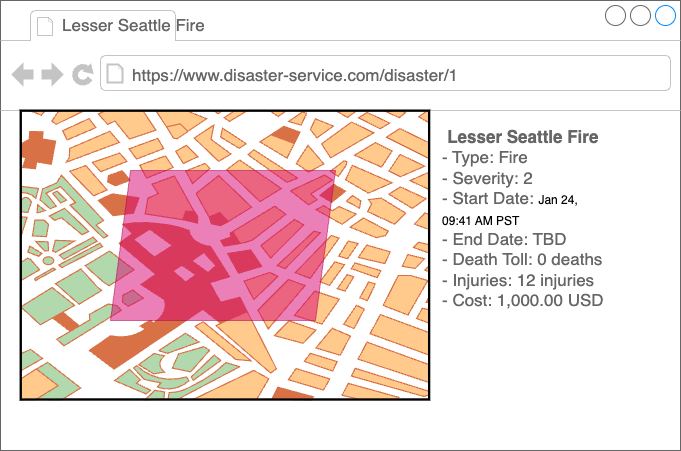
* Disaster history
  + Historical Records
  + Analytical views (Heatmaps)
* Providing disaster services resources
* Non-administrator user tracking
* Integration with natural disaster resources, such as the USGS and NWS
* Disaster forecasts
* Deleting natural disaster events
* Reporting erroneous natural 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?
3. Do we want to implement tiling of responses as part of MLP?
4. Disaster types – should we have different representations or properties for different disaster types? Do we need to create separate disaster tables?
5. Do we want to have create / update disaster pages, or only expose an API?

# 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

For the presentation layer, we propose using AWS Amplify. AWS Amplify is a fully-managed service for hosting server-side rendered and static frontend applications. It includes management of domains using Route53, as well as edge-location caching with AWS CloudFront. AWS Amplify also offers integration with popular git repository providers (GitHub, GitLab, and BitBucket) to offer Continuous Integration/Continuous Delivery deployments.

AWS Amplify also fits our specific use case, as it offers integration with Geospatial services, including integration with MapLibreGL for rendering maps, and Amazon Location Service for additional location services. AWS Amplify also offers integration with Amazon Cognito for implementing authentication and authorization, which includes provide authentication headers needed for integration with API Gateway.

## 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

Our backend service is relatively lightweight – we provide CRUD operations on Disasters, exposing a RESTful API. Our requests require minimal data analysis or transformation, and should complete in seconds. These requests are also stateless – we do not need to track user sessions or requests across sessions. We’d also like to avoid having to manage infrastructure.

AWS Lambda functions meet the requirements set by this criteria.  
  
AWS provides 1 million free AWS Lamba function requests each month, letting us stay within the free tier during development and in the early stages of adoption of the Natural Disaster Service.

There are two notable restrictions for AWS Lambda functions. One is a set timeout on function execution. Because we don’t need to perform compute intensive operations, this should not pose an issue.

A more likely issue to arise is a Lambda Function’s 10GB memory limit. If we need to render a large number of disasters, we may run out of memory during execution. This can be mitigated in several ways:

* Pagination of data
* Using separate requests for different disaster types
* Using map tiling features to only require data that fits within the bounding box of the current location and zoom level
* Reducing the fidelity of disaster event areas (such as using a circle or point) when rendering at higher zoom levels / larger tiles
  + This can be achieved using simplification of our geometry via PostGIS

We propose using a single AWS Lambda function for all API endpoints, and using the proxy integration with API Gateway. By using a single AWS Lambda Function, we reduce code duplication, aggregates logs for related APIs in a single source, and minimize the likelihood of cold starts due to lack of requests.

## 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 v2 with PostGIS extension

We propose using Amazon RDS (Relational Database Service) for our Disasters Database. We also propose using Amazon Aurora PostgreSQL Serverless as our database engine, and extending Aurora with the [PostGIS](https://postgis.net/) extension.

We propose using Aurora Serverless v2 to avoid needing to manage the underlying infrastructure of our database.

The PostGIS extension provides storage of spatial data, including GeoJSON compatible polygons. While not necessary for our MLP, PostGIS also provides spatial indexing, querying, and geometry processing (such as simplification).

The main limit to this solution is the requirement for a strict schema. While our MLP follows an identical schema for all disasters, if we need to included fields specific to different data types, we would need to create separate tables and schemas for each type. This would require revising our data model, creating new tables, and backfilling tables as needed, should we make this revision post-MLP

# Technologies

## Authentication

Cognito User Pool

We need to implement authentication and authorization to protect the creation and updating of disaster logs from non-trusted sources.

We propose using Cognito User Pools for authentication of our web application users. Cognito lets us sign up new users, as well as allow user sign in through a custom flow. We can then pass along identifier tokens in the form of the authorization header, for use by API Gateway and AWS Lambda. Cognito also lets us avoid needing to set up developer keys or IAM users for each administrator for the natural disaster service.

AWS Amplify has built-in support for Cognito User Pools, and API Gateway can use tokens for API authentication and authorization. We can also use Amplify Studio to simplify the development process.

## Frontend Framework

React

Amplify supports multiple different languages and frameworks for creating the web application, including Vanilla JavaScript, Angular, Next.js, Flutter, Vue, React Native, Android, and Swift. Which language and framework we use not only determines how to implement web application, but also which User Interface (UI) components are available.

We propose using the React library, as the Amplify React UI library provides UI components specifically for maps, as well as for authentication. By using React, we can also use Figma to make mocking and developing our UI easier.

## Map Rendering

MapLibre

We propose using MapLibre as the library for map rendering. Amplify has built-in support for MapLibre, as well as custom integrations between MapLibre and AWS Location Services.

MapLibre supports loading and rendering data from GeoJSON FeatureCollections, which we propose to use for our API models. If needed, MapLibre also supports vector tiling sources, if we opt to use tiling to improve performance. MapLibre can also be extended for additional use cases via third-party plugins.

## Backend Framework

Powertools for AWS Lambda (Python)

We propose using Powertools for AWS Lambda as a framework to implement our AWS Lambda Function. Powertools for AWS Lambda simplifies the process of managing and routing RESTful requests from API Gateway.

Powertools for AWS Lambda also simplifies working with metrics and logging of requests, making it easier to monitor the performance of our service. We can use provided middleware for authentication and authorization, and Pydantic for Data Validation of both request and response objects.

# Backend Service Design

## Model & Components

Class diagram

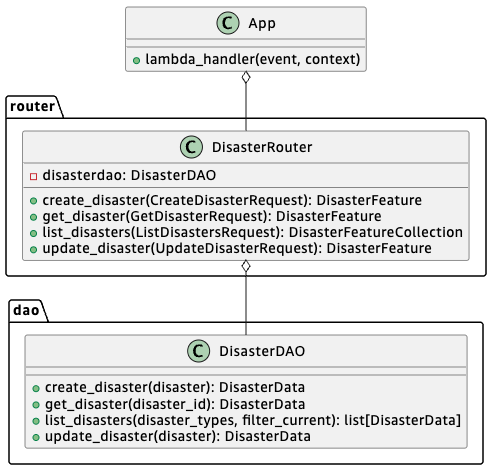
Disaster Data Class:
property properties: Disaster Data Properties
property area: Disaster Area Data

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

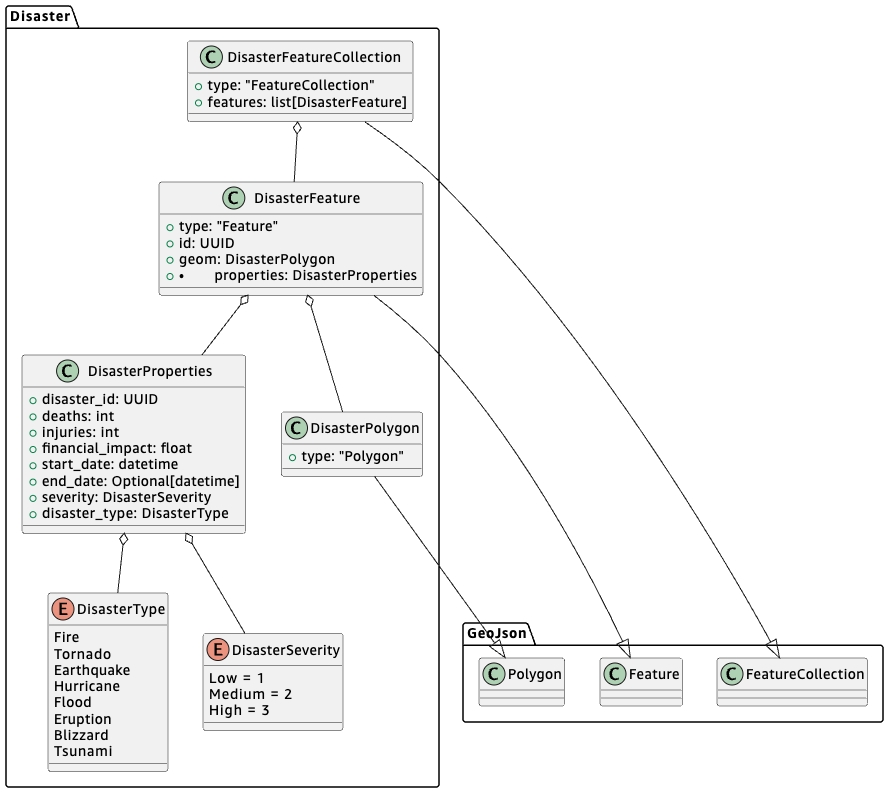
Disaster Area Data Class:
property Coordinates: list of points

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

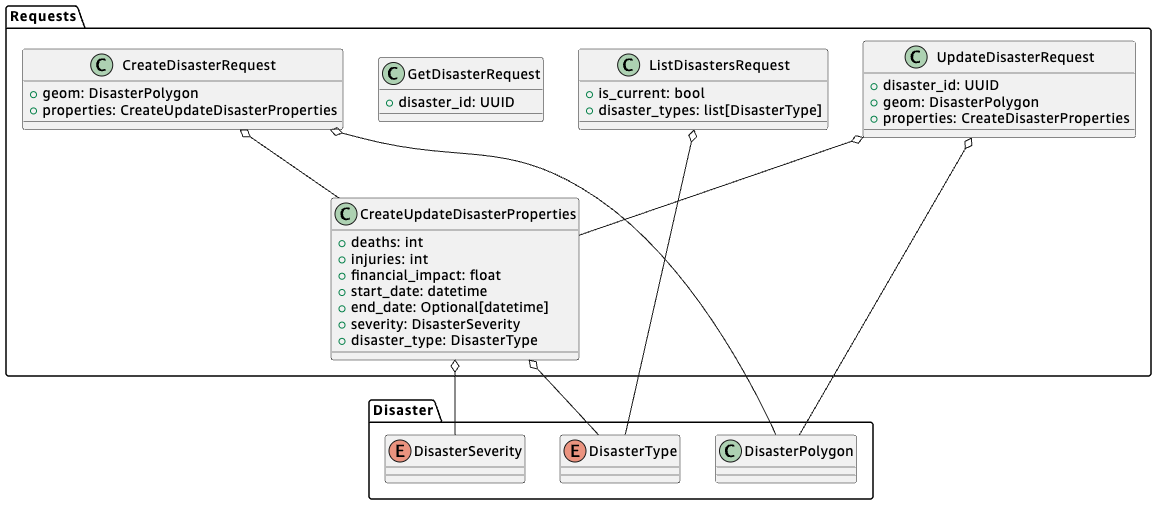
## API Design

### Public Models

#### Disaster

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

#### Requests



### Endpoints

#### CreateDisaster

Method:

* POST

Path:

* /disaster

Headers:

* Authorization

Request:

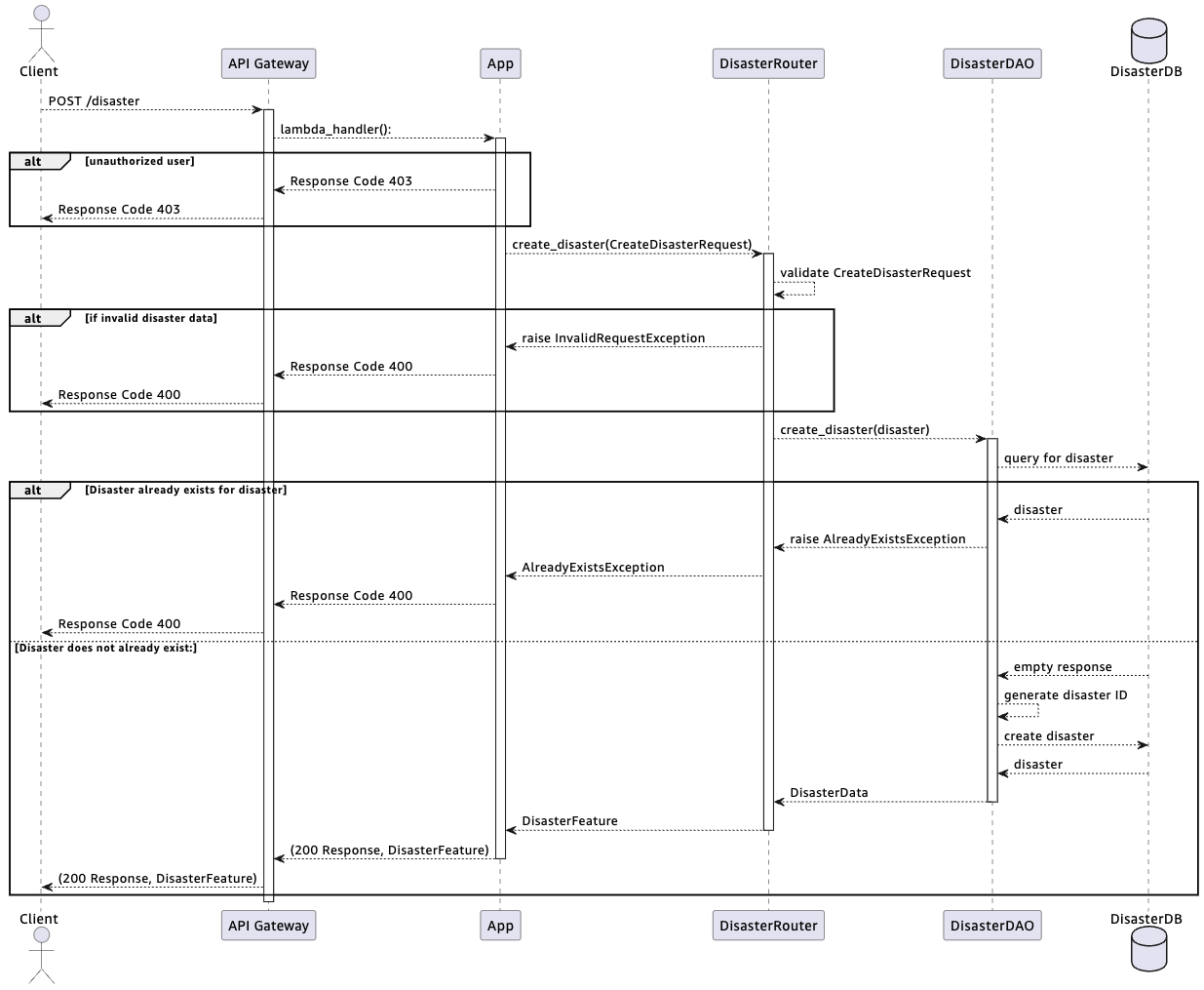
* CreateDisasterRequest (body)

Response:

* Status: 200
* DisasterFeature

Error Responses:

* 401: unauthorized request
* 500: internal server error



#### GetDisaster

Method:

* GET

Path:

* /disaster/{disaster\_id}

Request:

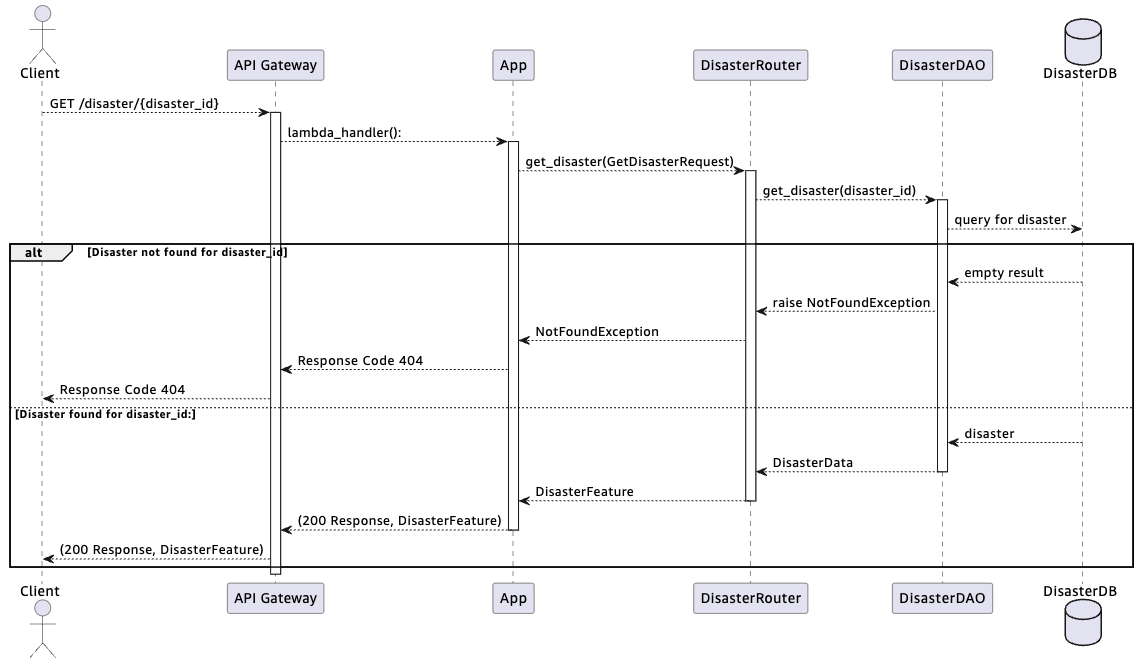
* GetDisasterRequest
  + {disaster\_id} -> disaster\_id

Response:

* Status: 200
* DisasterFeature

Error Responses:

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



#### ListDisasters

Method:

* GET

Path:

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

Request:

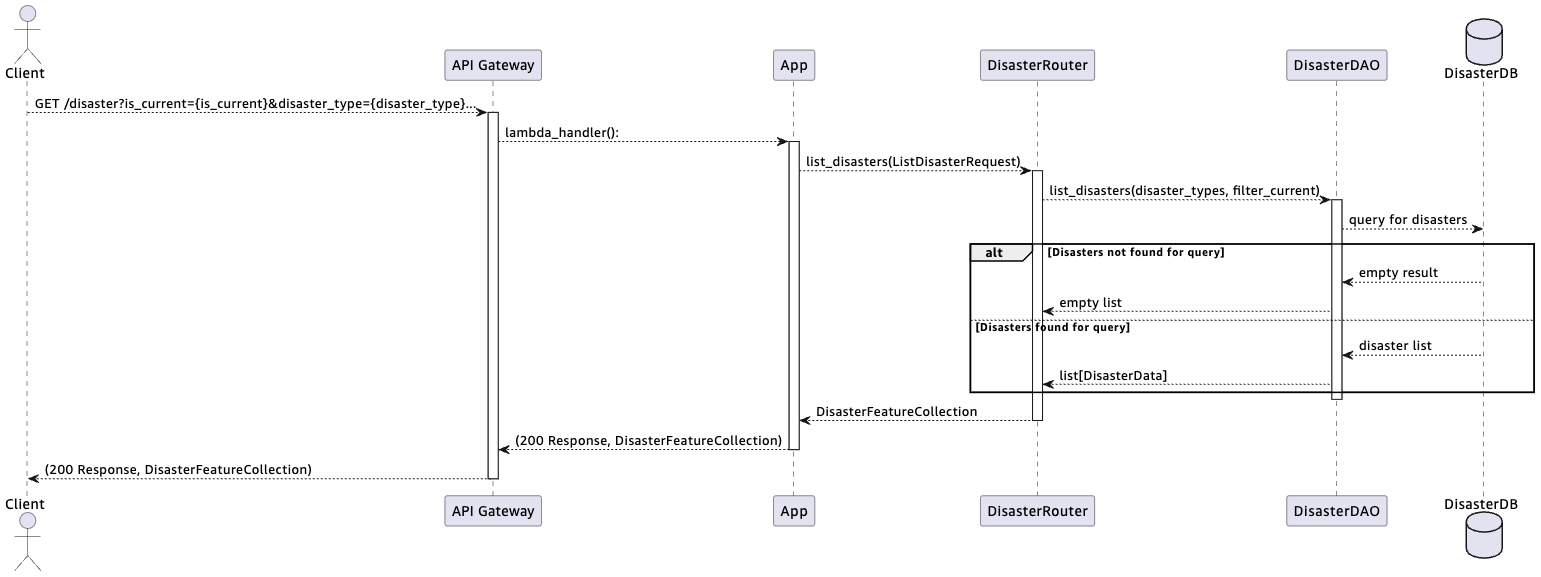
* ListDisastersRequest
  + {is\_current} -> is\_current
  + {disaster\_type} -> disaster\_types
    - Multiple: Collect into a list

Response:

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

Error Responses:

* 500: internal server error



#### UpdateDisaster

Method:

* PUT

Path:

* /disaster/{disaster\_id}

Headers:

* Authorization

Path parameters:

* disaster\_id: str (UUID)

Body:

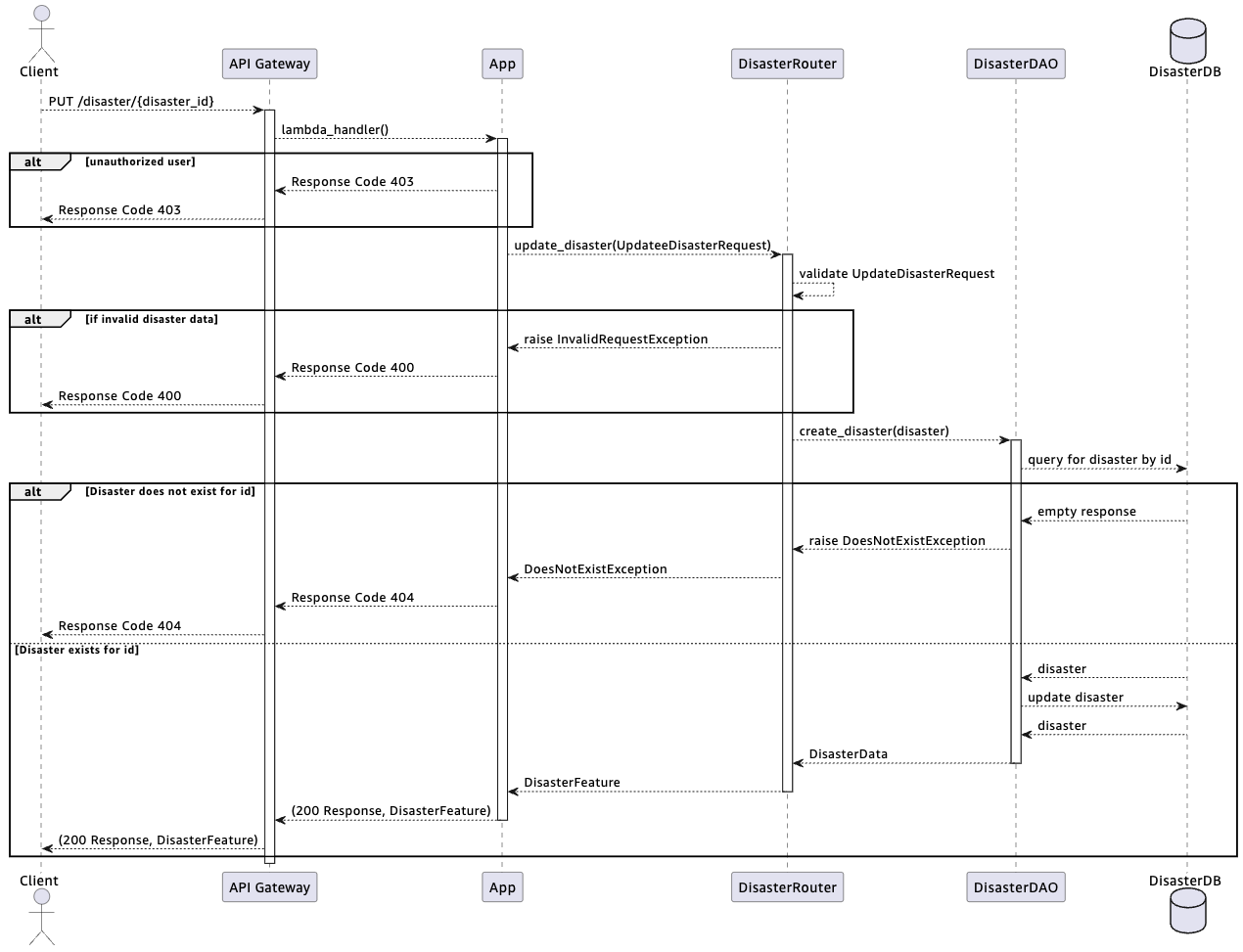
* UpdateDisasterRequest
  + {disaster\_id} -> disaster\_id
  + Body -> geom, properties

Response:

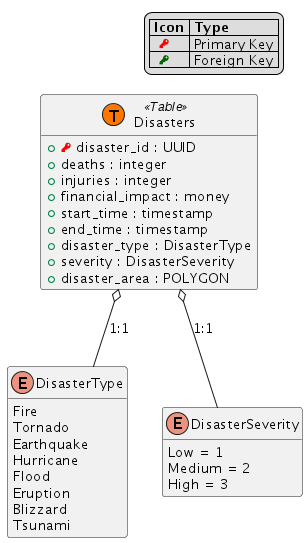
* Status: 200
* DisasterFeature

Error Responses:

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



# Data Design



# Tasks and Milestones

## Timeline

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Milestone | Status | Estimated Size (T-shirt Size, (Developer Days)) | Due Date (EOW = End of Week) | Date Started | Date Completed |
| 0 – Design | In Progress | L (10) | <EOW 2> | -- | -- |
| 1 – Infrastructure Setup | Not Started | M (5) | <EOW 3> | -- | -- |
| 2 – Common Components | Not Started | XL (15) | <EOW 6> | -- | -- |
| 3 – Disaster Details Page | Not Started | S (2.5) | <EOW 7> | -- | -- |
| 4 – Disasters Map Page | Not Started | S (2.5) | <EOW 7> | -- | -- |
| 5 – Create Disaster Page | Not Started | M (5) | <EOW 8> | -- | -- |
| 6 – Update Disaster Page | Not Started | M (5) | <EOW 9> | -- | -- |
|  | On Track | 45 Developer days (9 weeks) | <EOW 9> |  |  |

## Milestones

### Design

Milestone 0

##### Task 0.1

Create Design Document

##### Dependencies

* None

##### Success Criteria

* Created design document

##### Task 0.2

Design Review

##### Dependencies

* Task 0.1

##### Success Criteria

* Review Design Document with appropriate stakeholders
* Obtain sign-off to start implementation phase

### Infrastructure Setup

Milestone 1

#### Task 1.1

Create CDK package for Backend Service

##### Subtask 1.1.1

Initialize CDK application for backend service

###### Dependencies

* None

###### Success Criteria

* Git repository created for Disaster Backend Service
* Initial CDK template created and saved in git repository (`cdk init`)
* Deployed initial stack to CDK

##### Subtask 1.1.2

Initialize Backend Service Lambda Function

###### Dependencies

* Subtask 1.1.1

###### Success Criteria

* Lambda handler code (“Hello World”) created using Powertools for AWS Lambda
* Created Lambda Function & APIGateway `LambdaRestAPI` in CDK stack
* Deployed initial Lambda Function & API to CDK stack

##### Subtask 1.1.3

Initialize Disasters Database

###### Dependencies

* Subtask 1.1.1

###### Success Criteria

* Created Aurora Serverless v2 PostgreSQL database in CDK Stack
* Granted AWS Lambda appropriate permissions to access Disasters Database in CDK stack
* Added networking infrastructure to connect AWS Lambda to Disasters Database in CDK stack
* Deployed updated CDK Stack

#### Task 1.2

Create Amplify Application for Disaster Frontend Service

##### Subtask 1.2.1

Initialize Amplify Application

###### Dependencies

* None

###### Success Criteria

* Created git repository for Disaster Frontend Service
* Initialized Amplify Application code in git repository
* Deployed initial Amplify application code from git repository

### Common Components

Milestone 2

#### Task 2.1

Create Disasters Table in Disasters Database

##### Subtask 2.1.1

Investigate PostGIS extension

###### Dependencies

* None

###### Success Criteria

* Document how to set up PostGIS extension in Disasters Database in Subtask 2.1.2

##### Subtask 2.1.2

Set up PostGIS extension

###### Dependencies

* Subtask 1.1.3
* Subtask 2.1.1

###### Success Criteria

* PostGIS extension is set up in Disasters Database
* TODO: Add details based on subtask 2.1.2

##### Subtask 2.1.3

Create Disasters Table in Disasters Database

###### Dependencies

* Subtask 2.1.2

###### Success Criteria

* Created Disasters Table in Disasters Database in CDK stack
* Deployed updated CDK stack

#### Task 2.2

Common Disaster Backend Service Components

##### Subtask 2.2.1

Create DisasterDAO

###### Dependencies

* Subtask 1.1.2

###### Success Criteria

* Created stubs for DisasterDAO methods, returning mock data

##### Subtask 2.2.2

Create DisasterRouter

###### Dependencies

* Subtask 2.2.1

###### Success Criteria

* Created DisasterRouter module
* Created DisasterRouter methods, returning mock data

##### Subtask 2.2.3

Create Disaster APIs

###### Dependencies

* Subtask 2.2.2

###### Success Criteria

* Added API endpoints to APIGateway API
* Validated API endpoints return mock data from DisasterRouter methods

##### Subtask 2.2.4

Investigate API Authentication

###### Dependencies

* None

###### Success Criteria

* Document how to enforce authentication/authorization using Cognito for API Gateway in Subtask 2.2.5

##### Subtask 2.2.5

Implement API Authentication

###### Dependencies

* Subtask 2.2.3
* Subtask 2.2.4

###### Success Criteria

* Implemented authentication and authorization for Create / Update Disaster APIs
* TODO: Add details based on subtask 2.2.4

#### Task 2.3

Integrate Disaster Frontend and Backend Services

##### Subtask 2.3.1

Create Disaster Backend Service Client

###### Dependencies

* Subtask 1.1.2
* Subtask 1.2.1

###### Success Criteria

* Created API module in Disaster Frontend Service
* Created API client for Disaster Backend Service

##### Subtask 2.3.2

Investigate Amplify Authentication with Cognito

###### Dependencies

* Subtask 2.2.4

###### Success Criteria

* Investigated Amplify Authentication with Cognito and added details to subtask 2.3.3

##### Subtask 2.3.3

Implement Amplify Authentication with Cognito

###### Dependencies

* Subtask 2.2.5
* Subtask 2.3.2

###### Success Criteria

* Implement Amplify Authentication with Cognito
* Integrate Authentication with Disaster Backend Service API for Create / Update Disaster APIs
* TODO: Add details based on subtask 2.3.2

##### Subtask 2.3.4

Investigate rendering a map using MapLibre GL JS

###### Dependencies

* None

###### Success Criteria

* Investigate how to load and render polygons using MapLibre GL JS
* Investigate how to add popups with formatted content and links to polygons MapLibre GL JS
* Investigate how to draw polygons on map and add to submission form
* Add details to relevant subtasks in Milestones 3, 4, 5, and 6

### Disaster Details Page

Milestone 3

#### Task 3.1

Create GetDisaster API

##### Subtask 3.1.1

Implement `DisasterDAO::get\_disaster`

###### Dependencies

* Subtask 2.2.3

###### Success Criteria

* Query disaster table for disaster datum by id
* Raise `NotFoundException` if disaster not found for id
* Translate disaster datum to internal model and return

##### Subtask 3.1.2

Implement `DisasterRouter::get\_disaster`

###### Dependencies

* Subtask 3.1.1

###### Success Criteria

* Call `DisasterDAO::get\_disaster` for disaster datum
* Return 404 error if `NotFoundException` raised
* Translate disaster datum to public model and return

#### Task 3.2

Disaster Details Page

##### Subtask 3.2.1

Create Page for Disaster Details

###### Dependencies

* Subtask 2.2.3
* Subtask 2.3.1
* Subtask 2.3.4

###### Success Criteria

* Load Disaster datum from GetDisaster API
* Render disaster properties table on Disaster Details page
* Render disaster area polygon on map on Disaster Details Page

### Disasters Map Page

Milestone 4

#### Task 4.1

Create ListDisasters API

##### Subtask 4.1.1

Implement `DisasterDAO::list\_disasters`

###### Dependencies

* Subtask 2.2.3

###### Success Criteria

* Query disaster table for disasters, filtering for end\_time and disaster\_type as needed
* Translate disasters data to internal model and return

##### Subtask 4.1.2

Implement `DisasterRouter::list\_disasters`

###### Dependencies

* Subtask 4.1.1

###### Success Criteria

* Validate query parameters
* Retrieve disasters data from `DisasterDAO::list\_disasters`
* Translate disasters data to public model and return

#### Task 4.2

Disasters Map Page

##### Subtask 4.2.1

Create Page for Disasters Map

###### Dependencies

* Subtask 2.2.3
* Subtask 2.3.1
* Subtask 2.3.4

###### Success Criteria

* Load Disasters data from ListDisasters API
* Render disaster area polygons on map in Disasters Map page
* Add popups to each disaster area with summarized details and link to related Disasters Details Page

### Create Disaster

Milestone 5

#### Task 5.1

Create CreateDisaster API

##### Subtask 5.1.1

Implement `DisasterDAO::create\_disaster`

###### Dependencies

* Subtask 2.2.3

###### Success Criteria

* Translate internal model to data model for disaster datum
* Add disaster datum to Disasters Table
* Return internal model

##### Subtask 5.1.2

Implement `DisasterRoute::create\_disaster

###### Dependencies

* Subtask 2.2.5
* Subtask 4.1.1

###### Success Criteria

* Enforce authorization for CreateDisaster API
* Validate disaster datum provided in body
* Generate UUID for disaster
* Translate request to internal model
* Call `DisasterDAO::create\_disaster` with internal model
* Translate response from `DisasterDAO::create\_disaster` to public model and return

#### Task 5.2

Create Disaster Page

##### Subtask 5.2.1

Create Page for Create Disaster

###### Dependencies

* Subtask 2.2.3
* Subtask 2.3.1
* Subtask 2.3.3
* Subtask 2.3.4

###### Success Criteria

* Create form for Disaster Properties
* Add map and allow creation of polygon using map
* Submit form and polygon to CreateDisaster API

### Update Disaster

Milestone 6

#### Task 6.1

Create UpdateDisaster API

##### Subtask 6.1.1

Implement `DisasterDAO::update\_disaster`

###### Dependencies

* Subtask 2.2.3

###### Success Criteria

* Translate internal model to disaster datum
* Update disaster datum in Disaster Table using disaster datum
* Translate updated disaster datum to internal model and return

##### Subtask 6.1.2

Implement UpdateDisaster API

###### Dependencies

* Subtask 2.2.5
* Subtask 3.1.1
* Subtask 6.1.1

###### Success Criteria

* Enforce authorization for UpdateDisaster API
* Validate disaster datum provided in body
* Call `DisasterDAO::get\_disaster` to get disaster for id
* Return 404 error if `NotFoundException` raised
* Call `DisasterDAO::update\_disaster` to update disaster
* Translate result from `DisasterDAO::update\_disaster` to public model and return

#### Task 6.2

Update Disaster Page

##### Subtask 6.2.1

Create Page for Update Disaster

###### Dependencies

* Subtask 2.2.3
* Subtask 2.3.1
* Subtask 2.3.3
* Subtask 2.3.4

###### Success Criteria

* Create form for Disaster Properties
* Load data from GetDisasterAPI and pre-populate form
* Add map and allow update/creation of polygon using map
* Submit form and polygon to UpdateDisaster API

# Appendices

## Internal Documentation

* Requirements document: <Add link to requirements document here>

## 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. |
| UI | “User Interface” |
| 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” |

## Alternative Technologies

### Service Design

#### Django

One alternative to using the three-tier architecture, and combine our presentation tier and logic tier into a single application would be to use Django. Django follows a Model-View-Template pattern, using python to handle models and views, while using HTML templates to render user content.

Some advantage of Django is the ease of writing models and views in Python, the automatic creation of an administration console with authentication and authorization, integration with PostgreSQL databases, and the availability of third-party libraries to extend Django for Geospatial data.

On the other hand, Django’s design lends itself to a single, monolithic service for our web application. This means that the architecture would be strongly coupled by including rendering logic, business logic, and our data schema in one codebase. As our application grows, this could increase operational effort and make debugging and troubleshooting more difficult.

### Database

#### DynamoDB

We could use DynamoDB as an alternative to RDS and Aurora. DynamoDB is a highly-performant non-relational database.  
  
The advantages of DynamoDB would be increased performance, and a more flexible schema for our attributes. If needed, we could add attributes to different disaster logs based on disaster type, such as magnitude for earthquakes, categories for hurricanes and tornadoes, etc. Our disaster properties also lend themselves to the kind of simple queries DynamoDB excels at.

DynamoDB, however, does not support geospatial queries, and we would need to manually serialize and deserialize our geometric data. We also don’t know how many points our polygons will have, which may cause issues when querying large sets of data, due to the size restriction of DynamoDB query and scan results.

In addition, if our users are using our service to track live updates to disasters near them, we want our updates to be reflected immediately on a page refresh. While DynamoDB does offer optional strong consistency for queries, that imposes additional costs in terms of Read Capacity Units.

#### Redis Caching

Another consideration for our data is to use caching. Redis offers some functional for geospatial data, including spatial querying and serialization. We could use Redis on top of our database solution (relational or non-relational) to add this functionality and reduce latency.  
  
Due to the potentially large datasets that will be queried, and an unknown size of our geospatial objects (due to having an unknown number of points in each polygon), we run the risk of running out of space for our cached data. This solution also does not mitigate other downsides to using DynamoDB as our database solution, such as size restrictions on result lists.

In addition, if our users are using our service to track live updates to disasters near them, we want our updates to be reflected immediately on a page refresh. By caching, we risk supplying stale data.

### Map Rendering

#### Leaftlet

Leaflet is an open-source JavaScript library for mobile-friendly interactive maps. Leaflet offers many of the same features as MapLibre, and is easily extensible using third-party plugins.

We propose using MapLibre over Leaflet due to AWS Amplify’s built-in support for MapLibre, as well as available extensions for MapLibre using AWS Location Service and Amplify Geo.

## External Documentation

|  |  |
| --- | --- |
| Description | Link |
| Amazon Aurora - Managing spatial data with the PostGIS extension | [link](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Appendix.PostgreSQL.CommonDBATasks.PostGIS.html) |
| AWS Amplify API (REST) Define Authorization Rules | [link](https://docs.amplify.aws/react/build-a-backend/add-aws-services/rest-api/customize-authz/) |
| AWS Amplify documentation | [link](https://docs.amplify.aws/) |
| AWS Amplify Geo | [link](https://docs.amplify.aws/react/build-a-backend/add-aws-services/geo/) |
| GeoJSON format specification | [link](https://datatracker.ietf.org/doc/html/rfc7946) |
| MapLibre GL JS documentation | [link](https://maplibre.org/maplibre-gl-js/docs/) |
| MapLibre GL JS example – Add multiple geometries from one GeoJSON source | [link](https://maplibre.org/maplibre-gl-js/docs/examples/multiple-geometries/) |
| PostGIS documentation | [link](https://postgis.net/) |
| Powertools for AWS Lambda (Python) documentation | [link](https://docs.powertools.aws.dev/lambda/python/latest/) |
| Powertools for AWS Lambda (Python) – Security Schemes | [link](https://docs.powertools.aws.dev/lambda/python/latest/core/event_handler/api_gateway/#security-schemes) |
| Powertools for AWS Lambda (Python) – Data validation | [link](https://docs.powertools.aws.dev/lambda/python/latest/core/event_handler/api_gateway/#data-validation) |