# AquaPath Backend Documentation

## Introduction

AquaPath is an application designed to facilitate various functionalities related to marine transportation, weather tracking, and news retrieval. This documentation outlines the key components and functionalities of the AquaPath backend.

### MongoDB Integration

MongoDB is utilized as the primary database solution for AquaPath backend. It offers a flexible and scalable NoSQL database solution, which is well-suited for handling large volumes of unstructured data typically associated with marine transportation and weather tracking applications.

The server.js file serves as the main entry point for the application, initializing the Express server and connecting to the MongoDB database. It plays a crucial role in setting up and configuring the application, handling various middlewares, and managing routes. The file also listens on the specified port, allowing the application to start and respond to incoming requests.

## Imports and Configuration:

- **Importing Dependencies**: The file starts by importing necessary packages such as express, body-parser, mongoose, cors, and dotenv. These packages are essential for setting up the server, handling request parsing, connecting to the database, managing cross-origin resource sharing (CORS), and loading environment variables.
- **Environment Variables**: The dotenv package is used to load environment variables from a .env file. This allows the application to manage sensitive information such as database connection URLs and port numbers in a secure and flexible manner.

## Express App Setup:

- **Create Express App**: An Express application (app) is initialized using express(), allowing the application to handle incoming requests.
- **Body-Parser Middleware**: The body-parser middleware is configured to parse incoming JSON data and URL-encoded data. This allows the application to handle data sent in request bodies.
- **CORS Middleware**: The cors middleware is configured to allow cross-origin requests from any origin ('*'). This enables the application to interact with frontend clients or other servers located on different domains.
- **Static File Serving**: The application serves static files from the public directory and maps the /public endpoint to this directory. This is useful for serving static assets such as images and stylesheets.

## Route Management:

- **Route Initialization**: The application's routes are initialized by requiring the index.js file from the routes directory and passing the Express app as an argument. This sets up the various routes for handling incoming requests.

## Server Initialization:

- **Port Configuration**: The server is configured to listen on the port specified in the environment variables (or the default port of 8080 if not specified).
- **Database Connection**: The application connects to the MongoDB database using the URL specified in the environment variables. The useNewUrlParser option is used for compatibility with the latest versions of MongoDB.
- **Starting the Server**: Once the server is configured and the database connection is established, the application starts listening on the specified port. A success message is logged to the console indicating that the server has started.

### Error Handling:

- **Error Handling for Database Connection**: The file includes error handling for the database connection using the mongoose.connection.on('open') event listener. If an error occurs during the connection process, it is logged to the console.
- **Try-Catch Blocks**: The file uses try-catch blocks around the initialization and connection process to catch and log any errors that might occur during server setup.

### Exporting the App:

- **Exporting the Express App**: The Express app is exported as a module so that it can be imported and used in other parts of the application as needed.

# Models:

This folder contains two files: savedLocation.js and users.js. These files define the data models for saved locations and users, respectively.

### savedLocation.js:

This file defines the Mongoose schema for saved locations. This schema contains information about saved journeys between two coordinates and includes various properties of the journey.

**Schema Definition:**

- **srcCoordinates**: The starting coordinates of the journey, represented as a string (e.g., "37.7749,-122.4194"). This field is required.
- **destCoordinates**: The destination coordinates of the journey, represented as a string (e.g., "34.0522,-118.2437"). This field is required.
- **blockIceCaps**: A boolean indicating whether the journey should block ice caps.
- **allowPanama**: A boolean indicating whether the journey should allow crossing the Panama Canal.
- **allowSuez**: A boolean indicating whether the journey should allow crossing the Suez Canal.

In addition to these fields, the schema also includes a nested Properties schema, which contains details about the journey:

- **distance**: The total distance of the journey (in kilometers or miles).
- **mode**: The mode of transportation (e.g., "car", "plane").
- **departure**: The departure date and time.
- **arrival**: The arrival date and time.
- **duration**: The total duration of the journey.

The savedLocation schema is then exported as a Mongoose model.

### users.js:

This file defines the Mongoose schema for users. It includes user-specific information such as username, email, password, and saved locations.

**Schema Definition:**

- **userName**: The username of the user. This field is required.
- **email**: The email address of the user. This field is required and must be unique.
- **password**: The password of the user. This field is optional, as the user may authenticate using a third-party service such as Google.
- **fromGoogle**: A boolean indicating whether the user signed in using a Google account. This field is required.
- **profileImage**: The URL of the user's profile image. This field is optional.
- **savedLocation**: An array of savedLocation objects, representing the user's saved locations. This array is defined by importing the savedLocation schema from savedLocation.js.

Finally, the schema is exported as a Mongoose model named "User".

# Routes:

This folder contains several route files that handle various aspects of the application's functionality. These routes define the endpoints that the application can handle and the controllers that manage requests and responses for these endpoints.

## index.js:

This file serves as the main entry point for the application's routes. It imports the various route files and assigns them to specific paths in the application.

## Path Assignments:

- **/sea**: Routes requests to the seaRoute file.
- **/port**: Routes requests to the portRoute file.
- **/news**: Routes requests to the newsRoute file.
- **/weather**: Routes requests to the weatherRoute file.
- **/user**: Routes requests to the userRoute file.

The module exports a function that takes an Express app instance (app) as an argument and sets up the routes.

## newsRoute.js:

This file handles routes related to news.

**Endpoints:**

- **/getNews**: Handles GET requests to retrieve news data. This endpoint is managed by the getNews method in the newsController controller.

## portRoute.js:

This file handles routes related to port data.

**Endpoints:**

- **/portData**: Handles GET requests to retrieve port data. This endpoint is managed by the getPortData method in the portController controller.

## seaRoute.js:

This file handles routes related to sea routes.

**Endpoints:**

- **/getRoute**: Handles POST requests to retrieve a sea route. This endpoint is managed by the getRoute method in the seaController controller.
- **savePath**: This endpoint is commented out in the code. It would handle POST requests to save a path, and it would be managed by a method in the seaController controller.

## userRoute.js:

This file handles routes related to user management.

**Endpoints:**

- **/signUp**: Handles POST requests for user sign-up. This endpoint is managed by the signUp method in the userController controller.
- **/login**: Handles POST requests for user login. This endpoint is managed by the login method in the userController controller.
- **/googleLogin**: Handles POST requests for Google login. This endpoint is managed by the googleSignUp method in the userController controller.
- **/saveRoute**: Handles PUT requests to save a user route. This endpoint is managed by the saveRoute method in the userController controller.

## weatherRoute.js:

This file handles routes related to weather data.

**Endpoints:**

- **/getWeather**: Handles POST requests to retrieve weather data. This endpoint is managed by the getWeather method in the weatherController controller.

## Controllers:

- News API: **newsController.js**

  This module interacts with the News API to fetch news related to marine and cargo topics.

  The module requires the newsApi package, which is responsible for making requests to the NewsAPI. Additionally, it utilizes the dotenv package to load environment variables.

  The main function of this module is to retrieve news articles from various sources based on specific criteria.

  getNews Function

  This function retrieves news articles related to marine or cargo topics from specified sources. It utilizes the NewsAPI's v2.everything method to perform the search.

- Port Data API: port**Controller.js**

  This module is responsible for retrieving port data from a GeoJSON file and transforming it as required.

  The module imports the readAndTransformGeoJSON function from the portServices module. Additionally, it defines the file path to the GeoJSON file containing port data.

  This function reads the GeoJSON file using the readAndTransformGeoJSON function. If the data is successfully retrieved, it sends a response with a status code of 200 along with the port data. If an error occurs during data retrieval, it sends a response with a status code of 200 along with an error message.

- Weather API: weather**Controller.js**

  This module interacts with a weather service to fetch weather data based on provided latitude, longitude, and parameters.

  The module imports the getWeatherService function from the weatherService module.

  This function handles incoming requests to retrieve weather data. It expects latitude, longitude, and parameters to be provided in the request body. It then calls the getWeatherService function with these parameters to fetch the weather data.

  If weather data is successfully retrieved, it sends a response with a status code of 200 along with the weather data. If no data is found, it sends a response with a status code of 200 along with a message indicating that no data was found.

- Sea Route API: sea**Controller.js**

This controller module interacts with a sea route API to retrieve route data, weather data, and CO2 tender data based on provided origin and destination coordinates.

The module imports necessary dependencies including the sea route SDK, environment variables, and a model for saved locations.

This function handles incoming requests to retrieve sea route data. It expects the origin and destination coordinates, and additional parameters, to be provided in the request body. It then uses the sea route SDK to fetch route data based on the provided parameters.

The function also handles weather data and CO2 tender data retrieval based on the fetched route data. If no route is found, it sends a response with a status code of 404 indicating that no route was found between the origin and destination. If an error occurs during the process, it sends a response with a status code of 500 indicating an internal server error.

There is another API (Save Path) for updating the path preferences of the routes searched. The user can save the preferences using this API and helps in reducing the time to select the preferred routes as they can be preloaded.

## Public API's Documentation:

### Sea Route Planning API (For More Details)

- The Sea Route Planning API provides functionality to retrieve routes between source and destination locations at sea. It returns route details such as distance, duration, and zones traversed.
- Endpoint
- Base URL: https://api.searoutes.com/route/v2/sea/{coordinates}/plan
- Request Parameters: coordinates, allowIceAreas, avoidHRA, avoidSeca, blockAreas
- GET: /route/v2/sea/{coordinates}/plan
- Response
- The response contains route details including distance, duration, and zones traversed.

### CO2 Statistics API (For More Details)

- The CO2 Statistics API provides CO2e emissions statistics for a given carrier on a specified port pair.
- It calculates emissions based on vessel services and itineraries, considering factors such as fuel consumption and distance traveled.
- Base URL: https://api.searoutes.com/co2/v2/plan
- Request Parameters: fromLocode, toLocode, carrierId, nContainers
- GET: /co2/v2/plan
- Response: The response contains CO2e emissions statistics including minimum, maximum, average, and standard deviation values. It also provides detailed emissions with TTW (tank-to-wheels) and WTT (well-to-tank) emissions.

### Weather Along Route API (For More Details)

- The Weather Along Route API allows users to retrieve weather information along a specified route.
- This can be achieved by posting the route obtained from the /route/v2/ endpoint or by forming a custom Geo JSON object containing departure, arrival, duration, and speed properties.
- Base URL: https://api.searoutes.com/weather/v2/track
- Request Parameters: GeoJSON object: A Feature Collection containing at least one Feature with departure, arrival, duration, and speed properties.
- POST:  /weather/v2/track
- Response: The response contains weather information at each point along the specified route.