National and Kapodistrian University of Athens

ΥΣ09 – Software Engineering

Project - Real-Time Vessel Monitoring Using AIS Data

Group 0
Γιαβρίδης Αθανάσιος – 1115202100022
Κάστα Ξενοφών – 1115202100058
Μάντζαρη Ασημίνα Λεωνιδία - 1115202100094
Πετροπούλου Κωνσταντίνα - 1115202100157
Πράππας Γεώργιος – 1115202100163
Σταθοπούλου Μαρία - 1115202100177

Department of Informatics & Telecommunications,
Spring Semester 2025

Table of Contents

1. Abstract

This following report presents the design and development of a fullstack web application for real-time marine vessel tracking for the Software Engineering course. The system enables visualization and monitoring of ship movements on an interactive map, using simulated AIS (Automatic Identification System) data to represent dynamic maritime activity.

The backend is developed using Spring Boot, exposing a RESTful API to serve vessel and positional data. Apache Kafka is employed as a message broker for processing incoming vessel position updates in real time, ensuring high throughput and decoupled data ingestion. Vessel information and historical tracking data are stored and managed in a MySQL relational database. On the client side, a React-based frontend provides a responsive interface, allowing users to browse, filter, and inspect vessel details seamlessly.

The application demonstrates the integration of event-driven architecture with modern web technologies to build a scalable and responsive marine traffic system. It also highlights the use of streaming data pipelines and persistent storage to support live maritime situational awareness.

2. System Architecture

2.1 Overview

The system is designed as a distributed fullstack application composed of four main components: a data producer, a backend service, a relational database, and a frontend interface. Simulated AIS vessel data is generated and sent to an Apache Kafka topic, acting as a real-time message stream. A Spring Boot service acts as a Kafka consumer, continuously processing these messages, extracting relevant vessel information (such as position, speed, and heading), and storing the structured data into a MySQL database. The backend also exposes a RESTful API, allowing the React frontend to query and retrieve vessel data based on user interactions. The frontend provides a dynamic, map-based interface where users can view live vessel positions, search by vessel name or ID, and inspect details about each ship. This modular architecture ensures scalability and responsiveness, with Kafka enabling real-time data ingestion and decoupling the producer from the core application logic.

2.2 Data Management

Upon inspection of the contents within the P1 folder from Zenodo, it was determined that many of the files contained data irrelevant to the project's scope or unnecessary for transmission via Kafka. As a result, only the following files were retained:

- MMSI Country Codes.csv
- Ship Types List.csv
- nari_static.csv
- Navigational Status.csv
- nari_dynamic.csv
- nari_dynamic_sar.csv

These files include all relevant static data concerning countries, vessel types, and navigational statuses, as well as time-stamped vessel position information. Files related to "aton" (aid-to-navigation) vessels were excluded, as these were not within the scope defined in the original Software Requirements Specification (SRS).

To populate the database with static information, the init method within the StaticDataService class is invoked. This method first checks whether any records already exist in the Country, ShipType, or NavigationalStatus tables. If none are found, the method proceeds to read data from the corresponding CSV files and insert it into the database.

The vessel data import process involves an additional preprocessing step. Specifically, the create_vessels.py script is executed to filter and process the data from nari_static.csv, generating a new file, vessels.csv, which is then used for database insertion.

As port information was not available in the provided Zenodo dataset, unique port names were extracted from the destination field within the nari_dynamic files. Supplementary port details were retrieved from the World Port Index (WPI) database. These were compiled into a new ports.csv file and processed in the same manner as the other static tables.

2.3 Kafka Producer

Dynamic data used for real-time vessel tracking is sourced from the nari_dynamic.csv, nari_dynamic_sar.csv, and nari_static.csv files. Due to the large volume of tracking data, it is not feasible to store all entries in the database during server initialization. To address this, Apache Kafka is employed to stream data incrementally.

A Kafka producer, implemented in the producer.py script, filters and sorts the dynamic data by timestamp before sending it in real time to the ais-data topic. A corresponding Kafka consumer, implemented within the InstanceService class and subscribed to the same topic, retrieves the data in JSON format for further processing.

3. Spring-Boot Backend

3.1 Kafka Consumer

Once the consumer receives a vessel update from the Kafka topic, it transmits the data to the frontend via a WebSocket connection—only if the vessel's name and type are valid. A Data Transfer Object (DTO) is used to encapsulate only the essential data required by the client, thereby minimizing overhead. After transmission, the vessel's data is also stored in the database to preserve tracking history.

3.2 User Management and Authentication

User management on the backend is implemented through a structured set of classes and components that handle persistence, logic, and API exposure. The UserEntity class maps the user table in the database, while the UserDto serves as a data transfer object to safely expose only the necessary user fields to the frontend or across service layers—preventing sensitive data, such as passwords, from being leaked. The UserRepo acts as the data access layer, interfacing with the database to fetch and persist user data. Business logic is encapsulated in the UserService class, which defines methods for common operations such as user creation, updates, and account deletion. The UserController serves as the entry point for HTTP requests related to user activity, handling RESTful API calls and returning structured responses.

The backend exposes several endpoints, including registration (POST /register), login (POST /login), profile updates (PUT /users/:id), email and password changes, and account deletion. Users are identified by their unique ID, and data retrieval is primarily done via the findById method. To support operations like email and password updates, specific DTOs (EmailChangeDto and PasswordChangeDto) have been introduced to ensure separation of concerns and enhance security. By default, all new users have the notificationsActive flag set to true, which can be adjusted later from the Settings page.

3.3 Search System Logic

Search functionality in the backend is handled through RESTful API endpoints that query vessels, ports, and user fleets based on user input. Depending on the context (main map, vessels page, or ports page), different filters are applied. For map-based searches, additional logic identifies which visible vessels match the fleet name or other parameters such as vessel type. The backend ensures that results returned match both the search string and any active filters applied by the user.

3.4 Fleet Management and Data Handling

Fleet data is modeled in the backend through the Fleet entity, which represents the "fleet" table in the database. Each fleet can include multiple vessels, and this many-to-many relationship is handled through the FleetHasShips join table. The primary key for this table is defined using the FleetShipKey composite key entity. This allows the backend to manage which ships belong to which fleet, along with their active status in that fleet.

The backend exposes endpoints to support operations such as creating new fleets, retrieving all fleets associated with a user, adding or removing ships from fleets, and toggling the active status of ships within a fleet. These operations are secured and only accessible to authenticated users. Validation and ownership checks ensure that users can only modify fleets they own.

3.5 Vessel and Port API

The backend system is developed using Spring Boot and is structured around two primary controllers: VesselController and PortController. These controllers expose a RESTful API that facilitates the management and retrieval of vessel and port data. The controllers interact with the data layer through the use of Spring Data JPA repositories, namely VesselRepo and PortRepo, enabling seamless communication with the database and reducing boilerplate code for common operations.

For vessels, the API supports a variety of endpoints that handle both general and specific use cases. The endpoint /api/vessels/all provides a complete list of all vessels, which can be useful for bulk operations or offline data processing. For more interactive applications, such as dynamic tables in the frontend, the /api/vessels endpoint supports pagination through Spring's Pageable interface. This allows clients to fetch vessels in chunks, improving performance and user experience. Individual vessels can be retrieved using their unique ID via the /api/vessels/{id} endpoint. Additionally, there is a search feature implemented through /api/vessels/byname/{name}, which allows for partial name matching and is also paginated to handle large datasets efficiently. Updating a vessel's type is made possible via a PUT request to /api/vessels/{id}/type, where the ship type is modified based on a JSON payload sent from the client.

The port controller provides similar capabilities for managing ports. The /api/ports/all endpoint returns all ports, whereas the /api/ports endpoint handles paginated retrieval. Clients can also fetch specific port entries using their ID through /api/ports/{id}, or search for ports by name using /api/ports/byname/{name}, which returns paginated results in the form of partial objects (typically just IDs and names). Both controllers are annotated with @CrossOrigin to enable frontend access via CORS from http://localhost:3000.

Overall, the backend is designed to be modular, scalable, and well-integrated with the frontend via clearly defined JSON-based APIs. The use of pagination and search functionality ensures efficiency, especially when dealing with large datasets.

3.6 Query API

The backend of the Help Section is implemented using Spring Boot and is designed to handle user-submitted queries efficiently while offering functionality for retrieval, search, and bulk deletion of queries. Central to this system is the QueryController class, which exposes several RESTful API endpoints under the base path /api/queries, enabling interaction between the frontend and the database.

The primary function of this controller is to manage the lifecycle of user queries submitted through the "Contact Us" form on the frontend. When a user fills out the form and clicks submit, a POST request is sent to /api/queries. This request carries a JSON payload containing the user's first name, last name, email, phone number, and their question. The controller receives this data, maps it to a Query entity, and then delegates the saving operation to the QueryService, which abstracts the logic for interacting with the database using a QueryRepo repository.

For administrative or internal use, the backend exposes a GET /api/queries/all endpoint that retrieves the complete list of all submitted queries in one go. Additionally, a more efficient and scalable endpoint, GET /api/queries, supports pagination via Spring Data’s Pageable interface. This is especially useful for situations where large volumes of user queries need to be displayed in chunks, such as in an admin dashboard with table views and pagination controls.

The system also includes a bulk deletion mechanism. Through the POST /api/queries/delbulk endpoint, clients can submit a list of query IDs to be deleted in a single operation. This reduces the overhead of multiple delete calls and allows for quick management of stored queries.

Moreover, the backend offers a search capability through the endpoint GET /api/queries/byname/{name}. This endpoint is designed to retrieve queries submitted by users with a matching name, again with pagination support. The controller uses the QueryRepo's custom method getIdByName(name, pageable) to

perform this lookup and return only relevant entries, which could be useful for targeted support, moderation, or analytics.

To ensure proper communication with the frontend during development, Cross-Origin Resource Sharing (CORS) is enabled specifically for requests coming from http://localhost:3000, allowing the frontend application to interact with this backend without cross-origin issues.

In summary, the backend for the Help Section is a well-structured REST API with endpoints tailored for query submission, retrieval (both full and paginated), name-based filtering, and bulk deletion. It demonstrates a clean separation of concerns through service and repository layers, and it is built to scale as the number of user inquiries grows over time.

3.7 Administrator Services API

The administration system is supported by RESTful endpoints built with Spring Boot, using controller classes for each domain (UserController, VesselController, QueryController). Each controller supports pagination, sorting, CRUD operations, and custom logic relevant to its domain.

For user management, the backend supports operations like banning/unbanning through a PUT /api/users/{id}/ban endpoint. When a new user is created, there is a hardcoded check in place: if the user's email is admin@shiprec.com, they are automatically granted administrator privileges. This is a deliberate simplification for development and testing purposes, allowing easy bootstrap of the admin role without additional infrastructure like role assignment UIs or access management.

In the fleet backend, vessels are stored with various attributes including their shipType. The admin can update this via the PUT /api/vessels/{id}/type endpoint, which expects a JSON body specifying the new ship type.

For queries, the backend stores submissions via a POST /api/queries endpoint. It also supports retrieving all messages (GET /api/queries/all), paginated access (GET /api/queries), searching by name (GET /api/queries/byname/{name}), and bulk deletion

through the POST /api/queries/delbulk endpoint. The same endpoint pattern is used for users and vessels where applicable, offering a consistent API structure.

Each backend controller communicates with a corresponding Service and Repository layer. Repositories leverage Spring Data JPA to simplify database operations, while services encapsulate business logic, ensuring that controllers remain focused on request handling. To enable frontend-backend integration during development, CORS configuration allows cross-origin requests from http://localhost:3000.

4. React Frontend

4.1 Interface Navigation

The frontend navigation is implemented through a dynamic side navigation bar (NavigationBar) that adjusts its content based on the user's authentication status and role (guest, registered user, or admin). The navbar remains collapsed by default and expands on hover, showing both icons and corresponding labels. Common tabs like "Live Map", "Vessels", and "Ports" are always displayed, while additional options vary depending on the user's state. Guest users see tabs like "Register" and "Sign In", whereas registered users gain access to "My Account", "Settings", "Help", and "Sign Out". Admins additionally see an "Admin Page" tab at the top. Each tab is associated with a unique icon and changes style dynamically when active. The navigation bar also switches logos between admin and standard user modes and provides custom logic for signing out, which clears the user state from local storage and redirects to the sign-in page. Navigation is handled imperatively using window.location.href for immediate redirection, and routes are mapped to individual components via React Router.

4.2 Map Interface

The map interface is implemented using the `Leaflet.js` library. Incoming vessel data received over the WebSocket connection is displayed on the map using `Marker` components. Additional vessel information is shown in a `Popup`, which appears when the user clicks on a vessel.

The interface supports features such as dark mode, zooming, searching, and filtering by vessel type. Furthermore, user privileges vary based on authentication level. Guests (non-registered users) do not have access to features such as area-of-interest creation or advanced filtering options available through the search bar, which are reserved for registered and administrative users.

4.3 User Experience and Access Control

On the client side, user interaction and session management are handled using React. React’s useState and useEffect hooks are used

to manage and persist session data in localStorage, including a one-day expiration timestamp to improve security and reduce server load. A user is redirected to the Sign In page if their session expires or if they attempt to access protected routes. Route protection is enforced through a custom ProtectedRoute component located in frontend/src/components/ProtectedRoute.jsx, and applied globally in main.jsx.

The registration and login pages are implemented in registerPage.jsx and signInPage.jsx, respectively. During registration, users must provide their email, password, first and last name, and country. Input validation is performed client-side using regular expressions to ensure correct email formats and secure passwords (minimum of 8 characters with both letters and numbers). After successful authentication, the user is redirected to the main map interface. For country selection, the country-list package is integrated with react-select, using the packages react-select and country-list, both installed via npm.

Once logged in, users can access their profile and settings through the “My Account” section. The profile view and edit functionalities are handled in myProfilePage.jsx and editProfilePage.jsx, where users can update personal information, including an optional phone number (validated via regex). Controlled components are used for form handling, with changes processed through handleFieldChange, and updates submitted using handleSubmit, which issues a PUT request to the backend.

The SettingsPage.jsx offers additional account management features. Users can change their email or password via pop-up modals (ChangeEmailPopup.jsx, ChangePasswordPopup.jsx). These actions require password confirmation and include UI features such as password visibility toggles. Account deletion is also handled from the Settings page, with a confirmation modal (DeleteAccountPopup.jsx) ensuring users understand the irreversible nature of the action. Upon deletion, the user is redirected to the Guest Map view. Additionally, users can toggle their notification preferences; the value of notificationsActive is updated in the backend accordingly.

All user-facing pages and components are styled using dedicated CSS files located under frontend/src/styles.

4.4 Search UI and Filter Controls

Search components are integrated into almost all main pages: **Map**, **Vessels**,**Ports** and **Admin Page** In the Vessels and Ports pages, search functionality is a direct name match. On the Map page, users are provided with a more advanced search bar that allows filtering across multiple categories including "Vessels", "Ports", and "My Fleets". When "My Fleets" is selected, only vessels that are part of the user’s fleet and are currently active on the map are shown.

React useState and conditional rendering are used to display results in real time. The selected filters are preserved across searches to provide a consistent and intuitive user experience.

4.5 Fleet Interaction and Management

Fleet creation and management are implemented through a series of components found in the Vessels tab, Vessel Details modal, and the "My Fleets" page (accessible via the user's account). A registered user can create new fleets, add or remove ships, and toggle their active status within the UI.

Fleet interactivity is handled using useState, useEffect, and Axios for communicating with the backend endpoints. When viewing a fleet, ships are listed with interactive elements allowing the user to manage them. Buttons are used for toggling activity or removing a vessel, with appropriate confirmation and visual feedback (e.g., icons like a bin for deletion). Validation is included to ensure unauthorized users (guests) do not see or interact with fleet-related features.

4.6 Interface for Vessels and Ports

The frontend is built using React and includes a reusable component named ItemLists, which is responsible for rendering dynamic, interactive tables of either vessels or ports. The component is parameterized by a type prop that determines whether it will fetch and display vessel or port data. This modularity

allows the same component to handle different data types while maintaining a consistent user interface and experience.

The component initiates data fetching on mount and whenever the current page or items-per-page changes. It constructs a dynamic request URL based on the selected type and pagination parameters and fetches the data from the backend using the fetch API. The received data is stored in the component’s state and rendered as an HTML table, with column headers generated dynamically based on the keys of the returned objects. To enhance readability, certain keys like wpi and mmsi are automatically displayed in uppercase, while others are capitalized.

Pagination is fully supported within the component. Users can navigate between pages using arrow buttons and select how many items to view per page from a dropdown menu. The pagination system also displays the current range of items being viewed out of the total. This provides clarity and control over large datasets without overwhelming the user.

The component also implements sorting by allowing users to click on column headers to sort data in ascending or descending order. The sort state is managed internally, and the data is sorted client-side before rendering. This feature adds another layer of interactivity to the table and helps users find relevant information quickly.

In the case of vessels, the component allows row selection using checkboxes. Users can select individual rows or toggle a global “Select All” mode. In the global mode, users can still manually deselect specific rows, giving them fine-grained control. These selections are tracked using local state, distinguishing between explicitly selected and deselected items even when global selection is active.

Each row in the table is also interactive; clicking on a row navigates the user to a detailed page for that specific item using React Router's navigate function. This enables smooth transitions between the list view and the detailed view of an individual vessel or port.

Styling is handled using CSS modules (myprofile.css and lists.css), and visual elements such as dropdown arrows and navigation icons

are integrated to provide a modern and responsive user interface. Overall, the frontend component is designed for flexibility, performance, and ease of use, ensuring a polished experience for end-users managing vessel and port data.

4.7 User Support

The Help Section is implemented in two React components: FAQ.js and Contact.js.

4.7.1 Frequently Asked Questions

This component provides a list of frequently asked questions with toggleable answers. Questions are displayed as clickable buttons, and when a user clicks a question, its corresponding answer slides open below it. This is handled with React’s useState to keep track of the currently active index. A small arrow icon rotates to indicate whether a question is open or closed, improving UX. The FAQ content is hardcoded for simplicity.

4.7.2 Contacting Administrators

This part features a form where users can submit inquiries directly to the support team. The form captures the user's name, email, phone number, and their question. Upon submission, a POST request is sent to the backend endpoint (/api/queries) using fetch. Feedback messages are shown dynamically based on the response status, and the form is cleared on successful submission. Basic validation is applied to required fields.

4.8 Administrators User Interface

The Admin Panel in this application provides a centralized interface for managing users, fleets (vessels), and user-submitted messages (queries). The component responsible for rendering this interface is AdminLists, a React component that dynamically adapts based on the type of content being managed, which can be "users", "vessels", or "queries". This versatility allows a single component to handle all administrative listings with pagination, sorting, selection, search filtering, deletion, and conditional content editing, offering a consistent experience across the three data domains.

In the Users section, administrators can view a paginated and sortable list of registered users. Each row includes user information except sensitive fields like passwords. A key feature here is the ability to ban or unban users through a toggle button. Only unregistered users (i.e., already banned users) can be selected and deleted via the bulk deletion system. There's also a select-all mechanism, which supports global selection with manual overrides via a deselect list, ensuring administrators retain fine-grained control over which users are affected.

In the Fleets section, the administrator sees a list of registered vessels. Each vessel row includes various attributes, including an editable shipType. This type can be modified via an inline <select> dropdown that offers options like "Cargo", "Tanker", "Passenger", and more. Upon selection, a PUT request is sent to the backend to update the ship type in real time. The fleet section is not deletable via the same admin tools, reflecting the different nature of vessel data versus user or query entries.

The Messages (Queries) section displays user-submitted queries from the "Contact Us" form. These messages are displayed in a tabular format with an additional “View Query” button in each row. When clicked, this opens a QueryPopup modal showing the full contents of the message. Sensitive fields or long text bodies are excluded from the table by default to keep the interface clean, with the popup used for detailed inspection.

Across all sections, pagination and per-page configuration are available, allowing the administrator to control how many items are visible at once. Sorting is implemented per column, and visual indicators show the current sort direction. All data fetching is dynamically done through REST API calls to the backend based on the type of content and the current pagination settings.

5. Build System

The build system is automated through a bash script named *run-project.sh* which orchestrates launching the different parts of the application in separate terminal windows. First, it opens a new GNOME terminal to start the Kafka services by running *run-kafka.sh* inside the Kafka directory, ensuring the messaging backbone is up. After a short 2-second pause to allow Kafka to initialize, it opens another terminal window to start the backend server using the Maven wrapper with the command *./mvnw spring-boot:run* inside the backend folder. Another 2-second delay follows before finally launching the frontend React application in a new terminal by running *npm start* inside the frontend directory. Each terminal session is kept open (exec bash) to allow live monitoring and interaction. This setup ensures all core components start in the correct order and run concurrently for development convenience.