TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

Patrones AIS de transporte: detección de irregularidades en redes complejas

Estudante: Diego Seoane González

Dirección: Laura Milagros Castro Souto

Carlos Pais Montes

A Coruña, September de 2023.

A los que habéis creído en mi

Agradecimientos

Gracias a mi familia y amigos el haberme apoyado en toda esta etapa y haber confiado en mi durante todo el proceso. Y gracias también a Carlos por su inestimable ayuda y confianza, sin ti no habría sido posible.

Resumen

Este proyecto busca crear una herramienta que ayude a investigadores y analistas a comprender y analizar las redes globales de transporte marítimo. Se pretende facilitar la visualización de la gran cantidad de movimientos de barcos entre puertos generando rutas simbólicas sobre un mapa geográfico. Además, se busca agilizar y automatizar la gestión de dichos datos implementando procesos de ingesta masiva, limpieza, modificación y actualización.

Durante el proyecto se emplearán tecnologías maduras y confiables como Java, React, MySql o RabbitMq, además de a una arquitectura de micro-servicios que prima el rendimiento y la escalabilidad. Todo esto, unido a un buen diseño y buenas prácticas logrará que el sistema desarrollado sea robusto y sencillo de ampliar y mantener.

Abstract

This project aims to create a tool to help researchers and analysts understand and analyse global maritime transport networks. The aim is to facilitate the visualisation of the large number of ship movements between ports by generating symbolic routes on a geographical map. In addition, it seeks to streamline and automate the management of this data by implementing mass ingestion, cleansing, modification and updating processes.

During the project, mature and reliable technologies such as Java, React, MySql or RabbitMq will be used, as well as a micro-services architecture that prioritises performance and scalability. All this, together with a good design and good practices will make the developed system robust and easy to extend and maintain.

Palabras clave:

- Transporte marítimo
- Redes complejas
- Rutas marítimas
- Java
- Microservicios
- Arquitectura Hexagonal
- Arquitectura Dirigida a Eventos

Keywords:

- Maritime transportation
- Complex networks
- Maritime routes
- Java
- Microservices
- Hexagonal Architecture
- Event Driven Architecture

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La creciente globalización que ha sufrido el mundo en las últimas décadas ha supuesto un incesante aumento del comercio internacional (figura 1.1) y, por consiguiente, de la importancia de la logística.

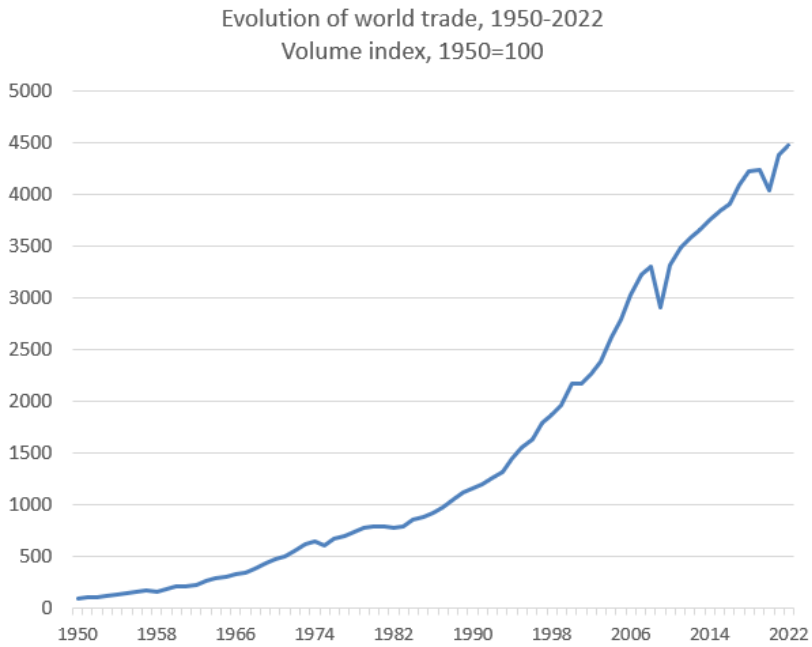


Figura 1.1: Gráfica de la OMC sobre la evolución del comercio

En este contexto, el medio marítimo es, sin lugar a dudas, el más dominante, gestionando - según datos de la UNCTAD[1] - más del 80% del volumen del comercio mundial. Esto conlleva que cualquier situación disruptiva como la crisis del COVID-19, el conflicto bélico en Ucrania o las crecientes guerras comerciales supongan grandes impactos tanto comerciales como económicos y políticos.

En un entorno tan crítico como es el de la logística marítima, tener la capacidad de com-

prender y analizar la situación global de las redes de transporte es crucial para ser capaces de tomar decisiones de altísima importancia, tanto para una gestión logística eficiente como en materia política, económica e incluso militar.

1.1 Motivación

Conociendo la importancia de la industria del transporte marítimo y sus grandes implicaciones en el mundo actual, se busca aportar algo distintivo para facilitar el análisis de las complejas redes de rutas y puertos.

Revisando el trabajo de Guerrero, Letrouit y Pais-Montes (2022) 'The container transport system during Covid-19: An analysis through the prism of complex networks'[2] y con la colaboración de uno de sus autores, Carlos Pais Montes, se pudo localizar una posibilidad avanzar y ayudar en futuras investigaciones: la gestión y visualización de ese complejo entramado de datos.

Tras un pequeño análisis, pudimos darnos cuenta de que la inclusión y análisis de nuevos datos suponía una repetitiva y costosa tarea de limpieza y organización de los mismos. Además nos encontramos con que la mayoría de las opciones para la visualización de estos datos son puramente matemáticas, siendo difícil hacer representaciones teniendo en cuenta la geografía mundial.

1.2 Alcance y objetivos

El objetivo principal de este proyecto es crear una aplicación que permita analizar datos sobre las redes de transporte marítimos de manera dinámica y proactiva. Para conseguir esto será necesario manejar grandes conjuntos de datos, limpiarlos de manera automática, ordenarlos de forma estructurada y, sobre todo, visualizarlos de forma rápida y amigable para poder sacar conclusiones fácilmente.

Todos estos datos serán proporcionados por sistemas AIS[1] y sería crucial disponibilizar dichos datos con la mayor inmediatez posible, facilitando de esta manera la detección irregularidades.

Es importante que las visualizaciones no sean simples muestras de información estáticas, si no que deberían tener la posibilidad de mostrar los datos acordes a ciertos criterios, haciendo de esta manera el análisis mucho más efectivo y específico. También sería deseable poder comparar distintos estados de la red en una misma visualización para así poder sacar conclusiones de mayor calado.

[1] Sistema de seguimiento automático utilizado por los servicios de trafico de buques que utiliza mediante transceptores en los propios buques

Además, es interesante tener la posibilidad no solo de analizar situaciones generales, si no también de obtener información acerca de elementos concretos como puertos o buques a lo largo del tiempo, conociendo así patrones sobre componentes concretos de toda la red de transporte.

Por desgracia, el contexto de este proyecto está acotado en al tratarse de un Trabajo de Fin de Grado, por lo que abarcar todas las necesidades e ideas aquí expuestas sería imposible. Es por esto por lo que se ha decidido desarrollar una aplicación básica que siente las bases para posibles desarrollos futuros, instaurando un sistema que permita integrar grandes conjuntos de datos, así como mostrarlos bajo dos enfoques básicos: el volumen gestionado por cada puerto a nivel global y la importancia de las rutas internacionales de transporte marítimo.

Capítulo 2
Metodología

En este capítulo se dará a conocer la metodología de desarrollo elegida para llevar a cabo este proyecto y cómo se ha adaptado para adecuarse a las necesidades y limitaciones del mismo.

Cuando hablamos de metodología de desarrollo de software, nos referimos al conjunto de herramientas y procesos que se utilizan para llevar a cabo un proyecto, desde su concepción y planificación hasta su mantenimiento. Estos marcos de trabajo nos ayudan a garantizar que el proyecto se complete de manera eficaz y eficiente, ayudándonos a evitar tanto errores como retrasos.

Existen muchas metodologías diferentes, cada una con sus puntos positivos y negativos, pero a grandes rasgos podemos dividirlas en dos grandes grupos con características similares:

2.0.1 Metodologías Tradicionales

Llevan más tiempo usándose y son las más formales, se caracterizan por seguir un enfoque secuencial en el que cada una de las fases del proyecto no se comienza hasta terminar la inmediatamente anterior. Algunas de las más importantes son la de Cascada, Prototipo y Espiral.

2.0.2 Metodologías Ágiles

Estas son bastante más recientes y mucho menos formales, siguen un enfoque iterativo en el que se desarrolla el proyecto en ciclos cortos aportando valor de forma más continuada. Algunos ejemplos serían XP, Scrum y Kanban.

En este proyecto se ha optado por utilizar Scrum[3], aunque con ciertas variaciones para adaptarlo a la situación del proyecto.

2.1 Scrum

Scrum apareció formalmente en 1995 y desde entonces es una de las metodologías con más adeptos y de las más usadas por empresas en todo el mundo sin importar su tamaño o especialización.

Scrum se basa en el desarrollo incremental para aportar valor de forma continua realizando entregas parciales del producto final a lo largo del tiempo. Cada uno de esos periodos es lo que se define en Scrum como Sprint.

Este enfoque es tremendamente útil en proyectos con requisitos vagamente definidos y que necesitan de una retroalimentación constante por parte del cliente.

2.1.1 Roles

Para formar un equipo en Scrum es necesario implementar tres roles básicos:

- **Product Owner**: es el responsable de definir y priorizar las historias de usuario a desarrollar en cada iteración.
- **Scrum Master**: es un director y facilitador, sus objetivos son tanto velar por el correcto funcionamiento de Scrum como trabajar por eliminar los obstáculos que se encuentren los miembros del equipo a la hora de desarrollar su trabajo.
- **Equipo**: es el conjunto de profesionales con distintas especialidades encargados de realizar las tareas seleccionadas para cada iteración.

2.1.2 Artefactos

En Scrum también se definen una serie de artefactos para organizar la información y que sea accesible para todos los miembros del equipo.

- **Historias de usuario**: es la unidad básica de trabajo en Scrum y describen una funcionalidad desde el punto de vista de un usuario. Son de gran utilidad para que los miembros del equipo sean capaces de comprender correctamente lo que requiere el usuario.
- **Product Backlog** (figura 2.3): es una lista priorizada de historias de usuario que debe mantener constantemente actualizada el Product Owner.
- **Sprint Backlog**: es el subconjunto de elementos del Product Backlog que se planean completar a lo largo de un Sprint.

2.1.3 Eventos

Además, Scrum proporciona una serie de eventos que ayudan a mantener la comunicación activa entre los miembros del equipo.

- **Sprint Planning**: es una reunión llevada a cabo al comienzo de cada Sprint para seleccionar y estimar los elementos del Sprint Backlog.
- **Daily Scrum**: reunión diaria en la que cada miembro del equipo actualiza la situación de su progreso con el fin de sincronizar a todos los miembros y poder gestionar posibles bloqueos.
- **Sprint Review**: esta reunión se realiza al final de cada Sprint con el objetivo de valorar el trabajo completado y obtener feedback de parte del Product Owner.
- **Sprint Retrospective**: esta reunión también se lleva a cabo al final del Sprint y su objetivo es valorar el funcionamiento del equipo y del proceso de trabajo para poder identificar y subsanar errores cometidos durante el Sprint.

2.2 Adaptación al Proyecto

Debido a la naturaleza del proyecto, en el que solo existe un miembro del equipo, es necesario realizar una serie de variaciones al modelo original de Scrum para adaptarlo a nuestra situación.

2.2.1 Elementos de Scrum

Para empezar, se eliminaría el rol de Scrum Master debido a que no existe ningún equipo que gestionar, y el rol de Product Owner sería desempeñado por el único miembro del equipo que haría a la vez de desarrollador.

En lo referente a las distintas reuniones, no tendría sentido realizar ni la Daily Scrum ni la Sprint Retrospective, ya que ambas tienen como principal función mejorar el trabajo del equipo, y en nuestro caso solo lo forma una persona.

El resto de reuniones se mantendrían, ya que es muy importante tanto organizar el trabajo a realizar en cada Sprint como valorar el trabajo realizado en el mismo.

2.2.2 Flexibilidad

Debido a que existe una única persona encargada de desarrollar la totalidad del proyecto, existe la posibilidad de ajustar la duración de cada Sprint de manera mucho más flexible.

De esta forma, podríamos encontrarnos diferencias en la duración de cada Sprint con el fin de desarrollar tareas similares y relacionadas en la misma iteración.

2.2.3 Comunicación con el Usuario

Debido a las características del proyecto, la interacción con el posible usuario final es bastante fluida. Por esta razón, se pone el foco en poder realizar entregas continuas incluso en medio del propio Sprint mediante prototipos, documentación o componentes incompletos. Todo esto nos permite obtener una rápida retroalimentación y ganar en mucha agilidad a la hora de realizar cambios o incluir nuevas funcionalidades.

2.2.4 Herramienta de Organización

Para poder gestionar y organizar todo el trabajo a realizar se utiliza una herramienta llamada *Notion*, su mayor característica es sin duda la flexibilidad lo cual nos ayuda a adaptar la propia herramienta y su uso a nuestras propias necesidades.

Las historias de usuario se han organizado en un tablón con tres bloques distintos: *Sin empezar*, *En Proceso* y *Completadas* (figura 2.1). Dichas funcionalidades se pueden asociar a un Sprint concreto por lo que es sencillo llevar una trazabilidad de las mismas y gestionar tanto el Product Backlog como el Sprint Backlog dependiendo si tienen asociado un Sprint concreto (figura 2.2).

Además, gracias a la flexibilidad de Notion, podemos añadir y gestionar todo el contenido de una Historia de Usuario de forma personalizada. De esta manera, podemos tener desgranadas las tareas a realizar, incluir una definición más completa de la funcionalidad e incluso enlazar documentos, archivos o referencias de posible interés. Esto nos permite tener un mayor control del trabajo a realizar y realizar una correcta documentación del trabajo a realizar.

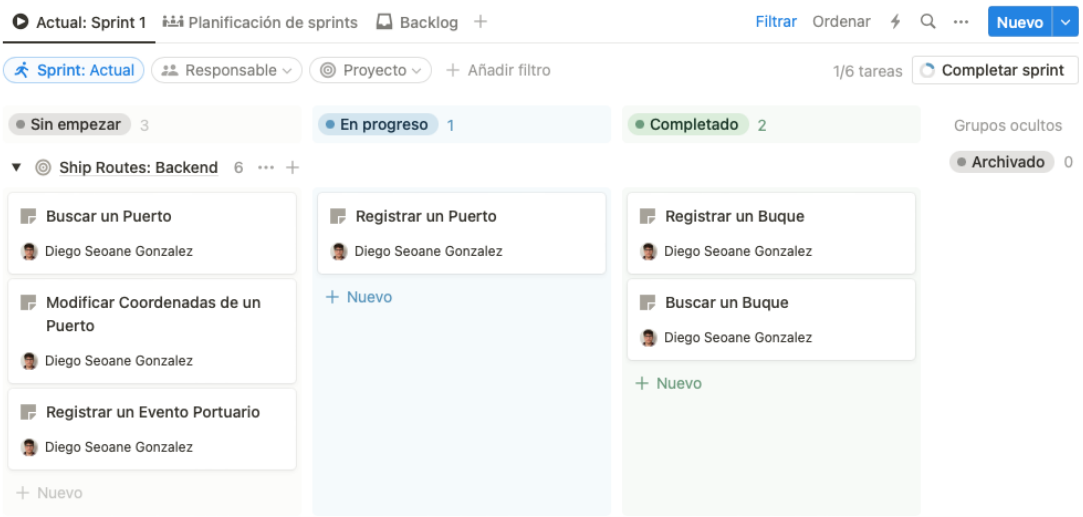


Figura 2.1: Ejemplo de tablón de un Sprint

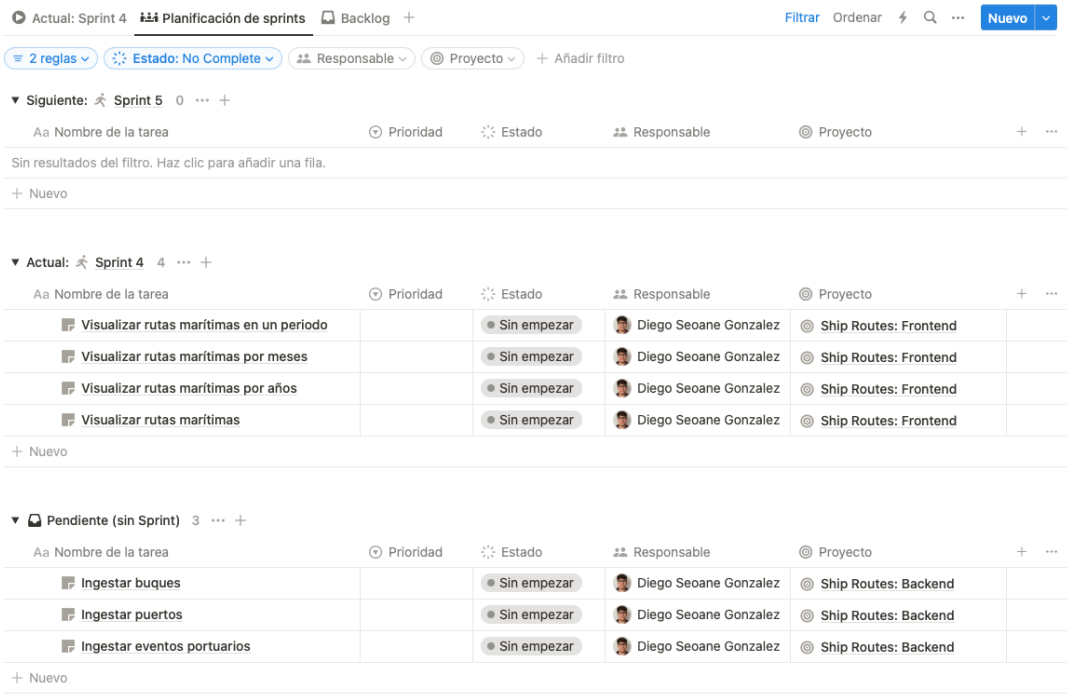


Figura 2.2: Ejemplo de organización de un Sprint

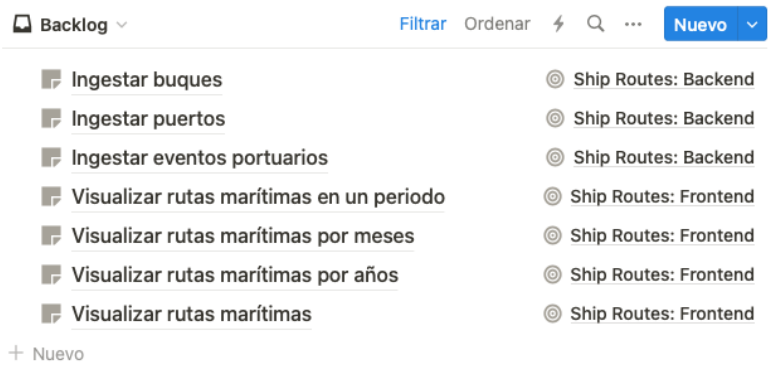


Figura 2.3: Ejemplo de situación del backlog

Capítulo 3

Análisis de requisitos

En este capítulo se expondrán los roles especificados en el sistema así como los requisitos tanto funcionales como no funcionales necesarios para obtener el resultado esperado. Por último se definirán las Historias de Usuario obtenidas de los diferentes requisitos.

3.1 Roles

El sistema contempla tres roles distintos, dependiendo del nivel de autorización del usuario que desea utilizar el servicio:

- **Usuario no autenticado**: Los usuarios no autenticados pueden acceder a vistas y datos básicos, además de tener la opción de registrarse o autenticarse.
- **Usuario autenticado**: Los usuarios autenticados, además de acceder a todas las vistas y datos básicos, tienen acceso a vistas más complejas y datos específicos y agregados.
- **Administrador**: Los administradores tienen el poder de consultar, registrar y modificar todo tipo de datos en el sistema. Además, tienen la capacidad de otorgar el rol de administrador a otros usuarios.

Es importante destacar que, debido a la posible inclusión de nuevas funcionalidades y considerando futuras necesidades de monetización del servicio, la cantidad de roles podría aumentar en el futuro.

3.2 Requisitos Funcionales

El sistema puede estructurarse en tres bloques clave: la visualización de los datos, la gestión de los mismos y la gestión de los usuarios. A continuación, se especificarán los requisitos funcionales específicos de cada unos de los bloques.

3.2.1 Visualización de Datos

El objetivo principal es proporcionar a los usuarios una visualización amigable de los datos disponibles sobre eventos portuarios y rutas de transporte. Esto se logrará mediante propiedades gráficas como tamaños y colores, lo que permitirá a los usuarios comprender rápidamente la importancia de los puertos y rutas sin necesidad de análisis numéricos exhaustivos.

Visualización de Eventos Portuarios

Sobre un mapa mundial, se mostrará la cantidad total de TEUS gestionados por los puertos registrados, agrupados por zonas geográficas. Esto permitirá a los usuarios obtener rápidamente una visión de los puertos y zonas más relevantes en el transporte marítimo de contenedores. Habrá cuatro vistas diferenciadas en función de la agregación temporal de los datos:

- Muestra de todos los datos como punto de entrada para nuevos usuarios.
- Agregación de los datos por año.
- Agregación por mes.
- Búsqueda en un intervalo de tiempo específico.

En las vistas agregadas por año y mes, se implementará una función de navegación que permitirá a los usuarios comparar y analizar rápidamente la situación entre periodos adyacentes.

Visualización de Rutas

Se mostrará la cantidad total de TEUS transportados en las rutas registradas, representando gráficamente el recorrido aproximado que realizaría un buque sobre un mapa del mundo. Las cuatro vistas definidas para la visualización de eventos portuarios también se aplicarán aquí.

3.2.2 Registro de Datos

El registro de datos se diseñará para permitir a los administradores incluir y modificar datos de manera eficiente. Se implementará principalmente a través de la ingesta masiva de datos, que se dividirá en tres conjuntos principales:

- Puertos: Incluye LOCODE, nombre y coordenadas.
- Buques: Incluye IMO, nombre y TEUS.

- Eventos portuarios: Requiere LOCODE del puerto, fecha del evento y tipo (entrada o salida).

Los trayectos entre puertos se podrán generar de manera automática basándose en los eventos portuarios, lo que permitirá el registro automático de las rutas. A mayores, el sistema debe ser capaz de generar el recorrido aproximado que realizaría un buque para cubrir dicha ruta.

Los datos en crudo de los eventos portuarios suelen encontrarse con información extendida de los puertos y buques, por lo que se podrán registrar los tres tipos de datos desde una misma ingesta.

Por desgracia, en ocasiones esos conjuntos de datos pueden tener ciertos errores, por lo que se habilitarán opciones para modificar a posteriori dichos datos. También es importante tener la opción de modificar el recorrido generado por el sistema para hacerlo más preciso.

Adicionalmente sería interesante dejar abierta la opción de en un futuro recibir esos datos a través de fuentes externas de manera automática.

3.2.3 Gestión de Usuarios

Al no ser la interacción con el usuario general la finalidad de este servicio, la gestión de los mismos será bastante sencilla, solamente se contará con tres operaciones a este respecto:

- Registro de un usuario con nombre de usuario, correo electrónico y contraseña.
- Autenticación.
- Promoción de un usuario a administrador, una operación que solo otro administrador puede llevar a cabo.

La seguridad del sistema se asegurará mediante la asignación de permisos adecuados a cada tipo de usuario.

3.3 Requisitos No Funcionales

De manera transversal, se prestará especial atención a los siguientes requisitos no funcionales:

- **Escalabilidad**: El sistema debe ser capaz de permitir un aumento significativo en la cantidad de datos y usuarios sin experimentar una degradación en el rendimiento. Esto es muy importante para garantizar que el sistema siga siendo eficiente a medida que crece.

- **Disponibilidad**: Debido a que ciertas operaciones como la ingesta son muy pesados, es esencial que el sistema esté preparado para ese aumento en la demanda de recursos y esté disponible para los usuarios de manera constante sin que afecte a otras operaciones.
- **Integridad de Datos**: La integridad de los datos es fundamental. Debe mantenerse en todo momento para garantiza la coherencia de la información que se muestra. Los datos deben reflejar de manera precisa la realidad de los eventos portuarios y las rutas de transporte marítimo.
- **Mantenimiento**: El sistema debe ser fácil de mantener y actualizar. Es importante que una vez esté en funcionamiento sea factible corregir posibles errores o añadir nuevas funcionalidades. Esto permitirá una gestión eficiente del sistema a lo largo del tiempo y la incorporación de nuevas formas de análisis de los datos almacenados.
- **Seguridad**: Se implementarán medidas de seguridad robustas para proteger los datos y garantizar que solo los usuarios autorizados tengan acceso a las funciones adecuadas del sistema.
- **Interoperabilidad**: El sistema debe ser capaz de interactuar y compartir datos de manera efectiva con otros sistemas y fuentes externas, lo que permitirá la integración con posibles fuentes de datos externas y la colaboración con otros sistemas relacionados. Esto es fundamental para garantizar que el sistema sea útil y eficaz en un entorno más amplio de intercambio de datos y colaboración entre diferentes aplicaciones y servicios.

3.4 Historias de usuario

A continuación se definirán las Historias de Usuario extraídas de los requisitos especificados anteriormente.

3.4.1 Visualización de datos

- **HU01 - Visualizar situación portuaria general:** Como usuario autenticado, quiero visualizar un mapa mundial que muestre la cantidad total de TEUS gestionados por los puertos registrados para comprender la importancia de las zonas geográficas y puertos en el transporte marítimo de contenedores.
- **HU02 - Visualizar situación portuaria por años:** Como usuario autenticado, quiero visualizar los datos de eventos portuarios agregados por año para realizar un análisis de la evolución de los puertos a lo largo del tiempo.

- **HU03 - Visualizar situación portuaria por meses:** Como usuario autenticado, quiero visualizar los datos de eventos portuarios agregados por mes para realizar un análisis más detallado de la situación portuaria a lo largo de los años.
- **HU04 - Visualizar situación portuaria en un periodo:** Como usuario autenticado, quiero visualizar los eventos portuarios en un intervalo de tiempo específico para analizar la situación portuaria en dicho periodo.
- **HU05 - Visualizar rutas marítimas:** Como usuario autenticado, quiero visualizar la cantidad total de TEUS transportados en las rutas registradas, representando gráficamente el recorrido aproximado que realizarían los buques, para comprender la importancia de cada ruta en el transporte marítimo de contenedores.
- **HU06 - Visualizar rutas marítimas por años:** Como usuario autenticado, quiero visualizar los datos de trayectos en las diferentes rutas agregados por año para analizar la evolución de las rutas de transporte marítimo a lo largo del tiempo.
- **HU07 - Visualizar rutas marítimas por meses:** Como usuario autenticado, quiero visualizar los datos de trayectos en las diferentes rutas agregados por mes para realizar un análisis más detallado de las rutas de transporte marítimo a lo largo de los años.
- **HU08 - Visualizar rutas marítimas en un periodo:** Como usuario autenticado, quiero visualizar los trayectos en las diferentes rutas en un intervalo de tiempo específico para analizar la situación del transporte marítimo en dicho periodo.

3.4.2 Registro de datos

- **HU09 - Ingestar Eventos Portuarios:** Como administrador, quiero registrar eventos portuarios, incluyendo LOCODE, IMO, fecha y tipo (entrada o salida), de forma masiva para mantener actualizada la información sobre los eventos portuarios.
- **HU10 - Ingestar Puertos:** Como administrador, quiero registrar información sobre puertos en el sistema, incluyendo LOCODE, nombre y coordenadas, a través de una ingesta masiva de datos para mantener actualizada la información sobre puertos.
- **HU11 - Ingestar Buques:** Como administrador, quiero registrar información sobre buques en el sistema, incluyendo IMO, nombre y TEUS, a través de una ingesta masiva de datos para mantener actualizada la información sobre buques.
- **HU12 - Generar Trayectos a partir de Eventos Portuarios:** Como administrador, quiero que el sistema genere automáticamente trayectos entre puertos basándose en los

eventos portuarios registrados para mantener actualizada la información sobre trayectos de transporte marítimo.

- **HU13 - Generar Rutas a partir de Trayectos:** Como administrador, deseo que el sistema genere automáticamente rutas de transporte marítimo de contenedores a partir de los trayectos registrados para mantener actualizada la información sobre rutas marítimas.
- **HU14 - Generar el Recorrido de una Ruta de Buques:** Como administrador, necesito que el sistema genere automáticamente el recorrido aproximado que seguiría un buque para cubrir una ruta específica de transporte para permitir una visualización más cómoda de las rutas.
- **HU15 - Registrar un Buque:** Como administrador, quiero registrar un nuevo buque en el sistema proporcionando información como IMO, nombre y TEUS para mantener actualizada la información de los buques.
- **HU16 - Buscar un Buque:** Como usuario autenticado, quiero buscar información sobre un buque específico en el sistema para recuperar su IMO, nombre y TEUS.
- **HU17 - Registrar un Puerto:** Como administrador, quiero registrar un nuevo puerto en el sistema proporcionando información como LOCODE, nombre y coordenadas para mantener actualizada la información de los puertos.
- **HU18 - Modificar Coordenadas de un Puerto:** Como administrador, quiero modificar las coordenadas de un puerto existente en el sistema para corregir errores o refinar la precisión de la ubicación del puerto.
- **HU19 - Buscar un Puerto:** Como usuario autenticado, quiero buscar información sobre un puerto específico en el sistema para recuperar su LOCODE, nombre, localización, número total de eventos portuarios registrados y número total de TEUS gestionados.
- **HU20 - Registrar un Evento Portuario:** Como administrador, quiero registrar un nuevo evento portuario en el sistema proporcionando detalles como LOCODE del puerto, fecha del evento y tipo (entrada o salida) para mantener actualizada la información de los eventos portuarios.
- **HU21 - Registrar una Ruta:** Como administrador, quiero registrar una nueva ruta de transporte marítimo de contenedores en el sistema para mantener actualizada la información sobre las rutas disponibles.

- **HU22 - Registrar un Trayecto:** Como administrador, quiero registrar un nuevo trayecto entre puertos en el sistema para mantener actualizada la información sobre los trayectos realizados por los buques.

3.4.3 Usuarios

- **HU23 - Registrarse:** Como usuario no autenticado, quiero tener la capacidad de registrarme en el sistema proporcionando un nombre de usuario, correo electrónico y contraseña para acceder a las funciones completas del sistema.
- **HU24 - Autenticarse:** Como usuario no autenticado, quiero iniciar sesión en el sistema utilizando mis credenciales (nombre de usuario y contraseña) para acceder a las funciones completas del sistema.
- **HU25 - Promocionar a administrador:** Como administrador, quiero tener la capacidad de ascender a un usuario existente al rol de administrador para que el usuario pueda llevar a cabo tareas de administración en el sistema.

Capítulo 4
Planificación

El proyecto se divide en 7 sprints, cada uno con objetivos concretos para desarrollar nuevas funcionalidades en la aplicación. Un enfoque fundamental a lo largo de todos los sprints es la revisión constante de los resultados anteriores, permitiendo realizar mejoras y correcciones según sea necesario.

En cuanto a las estimaciones, optaremos por una medida relativa (Puntos Historia o PH) en lugar de unidades de tiempo, esta elección se basa en la idea de que es más sencillo clasificar las Historias de Usuario según su tamaño que intentar determinar cuánto tiempo tomará cada una. Para lograr esta clasificación, seguiremos la secuencia de Fibonacci, que nos ayuda a distinguir elementos de mayor tamaño.

Este enfoque proporciona la flexibilidad necesaria para adaptar el sistema a medida que se acumula retroalimentación y se adquieren nuevos conocimientos.

4.0.1 Sprint 0

En este sprint inicial, el equipo se familiarizará con el dominio del transporte marítimo de contenedores, analizará los datos disponibles y establecerá el entorno de desarrollo.

Se estima una complejidad de **13 Puntos Historia**.

4.0.2 Sprint 1

En este sprint (cuadro 4.1), nos centraremos en el registro de datos sobre puertos, buques y eventos portuarios, así como en la capacidad de buscar información sobre puertos y buques.

Se estima una complejidad total de **31 Puntos Historia**.

4.0.3 Sprint 2

Este sprint (cuadro 4.2) se enfocará en la visualización de datos sobre puertos y eventos portuarios. Debido a que todas las visualizaciones tienen en común gran parte de su desarrollo,

está claro que la primera que se implemente requerirá mucho más trabajo que el resto.

Se estima una complejidad total de **37 Puntos Historia**.

4.0.4 Sprint 3

En este sprint (cuadro 4.3), se continuará con el registro de datos, permitiendo la creación de trayectos y rutas, así como la generación de recorrido de las mismas.

Se estima una complejidad total de **34 Puntos Historia**.

4.0.5 Sprint 4

Este sprint (cuadro 4.4) se enfocará en la visualización de datos sobre rutas marítimas y trayectos de buques. Al igual que en la visualización de la situación portuaria, la primera de las implementaciones requiere una mayor carga de trabajo.

Se estima una complejidad total de **32 Puntos Historia**.

4.0.6 Sprint 5

En este sprint (cuadro 4.5), se implementará la ingesta de datos.

Se estima una complejidad total de **31 Puntos Historia**.

4.0.7 Sprint 6

Este sprint (cuadro 4.6) se centrará en la gestión de usuarios.

Se estima una complejidad total de **18 Puntos Historia**.

4.0.8 Sprint 7

En este sprint final, se desplegará la aplicación para disponibilizarla y se documentarán todas las actividades realizadas durante el proyecto, incluyendo el análisis, diseño, implementación, pruebas y resultados obtenidos.

Se estima una complejidad de **8 Puntos Historia** la consecución del sistema y su despliegue.

4.1 Cálculo de costes

El cálculo de los costos del proyecto es esencial para una gestión financiera efectiva. En este proceso, consideramos varios elementos clave que contribuyen a la inversión total, a continuación desglosamos los costos en función de los mismos:

HU	Nombre	Estimación
HU15	Registrar un Buque	8 PH
HU16	Buscar un Buque	2 PH
HU17	Registrar un Puerto	8 PH
HU18	Modificar Coordenadas de un Puerto	5 PH
HU19	Buscar un Puerto	2 PH
HU20	Registrar un Evento Portuario	8 PH
TOTAL		31 PH

Cuadro 4.1: Historias de usuario y estimación del Sprint 1

HU	Nombre	Estimación
HU01	Visualizar situación portuaria general	21 PH
HU02	Visualizar situación portuaria por años	8 PH
HU03	Visualizar situación portuaria por meses	3 PH
HU04	Visualizar situación portuaria en un periodo	5 PH
TOTAL		37 PH

Cuadro 4.2: Historias de usuario y estimación del Sprint 2

HU	Nombre	Estimación
HU12	Generar Trayectos a partir de Eventos Portuarios	13 PH
HU13	Generar Rutas a partir de Trayectos	2 PH
HU14	Generar el Recorrido de una Ruta de Buques	13 PH
HU21	Registrar una Ruta	3 PH
HU22	Registrar un Trayecto	3 PH
	TOTAL	34 PH

Cuadro 4.3: Historias de usuario y estimación del Sprint 3

HU	Nombre	Estimación
HU05	Visualizar rutas marítimas	21 PH
HU06	Visualizar rutas marítimas por años	5 PH
HU07	Visualizar rutas marítimas por meses	3 PH
HU08	Visualizar rutas marítimas en un periodo	3 PH
TOTAL		32 PH

Cuadro 4.4: Historias de usuario y estimación del Sprint 4

HU	Nombre	Estimación
HU09	Ingestar Eventos Portuarios	21 PH
HU10	Ingestar Puertos	5 PH
HU11	Ingestar Buques	5 PH
TOTAL		31 PH

Cuadro 4.5: Historias de usuario y estimación del Sprint 5

HU	Nombre	Estimación
HU23	Registrarse	8 PH
HU24	Autenticarse	5 PH
HU25	Promocionar a administrador	5 PH
TOTAL		18 PH

Cuadro 4.6: Historias de usuario y estimación del Sprint 6

4.1.1 Costo del Desarrollador

El recurso humano es una parte fundamental del proyecto. Estimamos que el proyecto tendrá un total de 204 Puntos Historia en desarrollo. Basándonos en un rendimiento promedio de 45 Puntos Historia al mes por parte del desarrollador, proyectamos que el proyecto tendrá una duración de aproximadamente 4,5 meses.

Considerando que el desarrollador tiene una tarifa mensual de 1600€[1], calculamos el costo total del desarrollador como sigue:

- Costo mensual del desarrollador: 1600€
- Duración estimada del proyecto: 4,5 meses
- **Costo total del desarrollador**: 1600€ x 4,5 semanas = **7200€**

4.1.2 Costo del Equipo de Trabajo (Portátil)

La herramienta de trabajo esencial para el desarrollador es su portátil. Para este proyecto, se ha utilizado un MacBook Pro 13" con un costo de 1600€[2]. Se espera que este portátil tenga una vida útil de aproximadamente 6 años.

Dado que hay 12 meses en un año, podemos calcular el costo mensual del portátil como sigue:

- Costo del portátil: 1600€
- Vida útil estimada del portátil: 6 años = 72 meses
- Costo mensual del portátil: 1,600€ / 72 meses ≈ 22,22€
- **Costo total del portátil**: 22,22€ x 4,5 meses = **100€**

4.1.3 Costo de Internet

La conectividad a internet es esencial para el desarrollo y la colaboración en el proyecto. El costo mensual de internet es de 30€ al mes[3]. Durante el periodo estimado del proyecto, que es de aproximadamente 4,5 meses, el costo total de internet sería de:

- Costo mensual de Internet: 30€
- Duración estimada del proyecto: 4,5 meses
- **Costo total de Internet**: 30€ x 4,5 meses = **90€**

[1] Sueldo aproximado del autor
[2] Precio en el momento en el que el autor lo adquirió
[3] Precio de la tarifa del autor

4.1.4 Costo Total del Proyecto

Sumando los costos del desarrollador, el portátil e internet, obtenemos el costo total del proyecto:

- Costo total del desarrollador: 7200€
- Costo total del portátil: 100€
- Costo total de Internet: 90€
- **Costo total aproximado del proyecto**: 7200€ + 100€ + 90€ = **7390€**

Capítulo 5

Diseño

En este capítulo se darán a conocer las decisiones de diseño más importantes tomadas con el fin de desarrollar un sistema de la mayor calidad posible.

5.1 Objetivos y principios

Antes de adentrarnos en las decisiones de diseño específicas, es crucial establecer los objetivos que guiarán todo el proceso de desarrollo. A continuación se definen seis objetivos clave que buscamos cumplir:

- **Bajo Acoplamiento:** permite que las diferentes piezas del sistema funcionen de manera aislada, lo que facilita las modificaciones sin afectar al resto de componentes.
- **Cohesión:** los componentes cohesivos son más fáciles de entender, probar y mantener.
- **Testabilidad:** priorizar la testabilidad simplifica la detección y corrección de errores.
- **Flexibilidad:** el sistema debe ser capaz de adaptarse a requisitos futuros sin que esto implique un trabajo adicional sobre lo existente. Esto permite introducir nuevas características de manera eficiente.
- **Mantenibilidad:** facilita realizar cambios, mejoras y correcciones en el sistema, lo que permite reducir los costos y el tiempo requerido de futuras actualizaciones.

5.1.1 Principios SOLID

Para poder lograr esos objetivos de diseño a lo largo de todo el sistema, se ha decidido seguir los principios SOLID[4], un conjunto de cinco principios ampliamente aceptados en el desarrollo de software. Estos nos ayudarán a tener una guía sólida para tomar decisiones coherentes y desarrollar una solución de calidad.

Principio de Responsabilidad Única (SRP)

Este principio establece que un componente debe tener una única razón para cambiar o presentar un único concepto. Esto se consigue principalmente implementando clases pequeñas con objetivos muy acotados.

Seguir este principio nos aporta una alta cohesión y robustez, permite composición e inyección de colaboradores y además ayuda a evitar la duplicidad de código.

Ejemplo: una clase de servicio debería ejecutar una única acción y no implementar varios casos de uso en el mismo componente (figura 5.1).

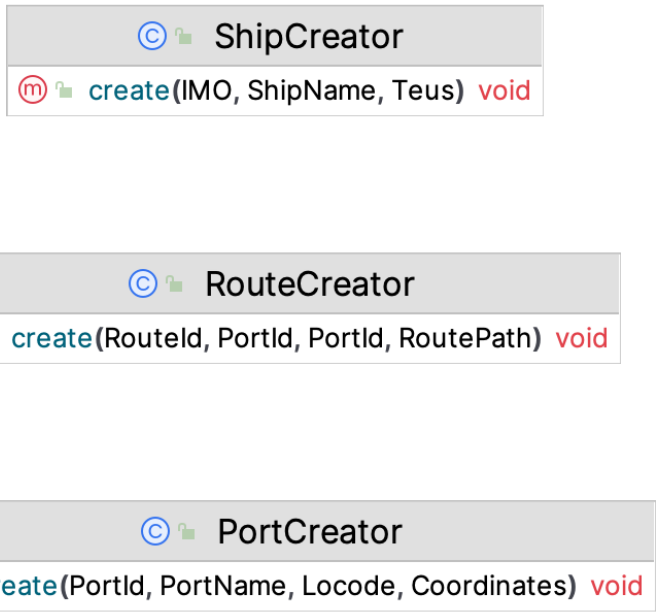


Figura 5.1: Ejemplo de uso del Principio de Responsabilidad Única (SRP)

Principio Abierto/Cerrado (OCP)

El OCP propone que las entidades del software deben estar abiertas para su extensión pero cerradas para su modificación. Esto se logra mediante interfaces y abstracciones evitando depender de implementaciones específicas.

Este principio facilita añadir nuevos casos de uso y funcionalidades aportando gran flexibilidad y a mayores simplifica la realización de los tests de estos componentes.

Ejemplo: un servicio no debería ocuparse de realizar las acciones derivadas de dicho caso de uso ya que estas pueden aumentar a lo largo del tiempo, para solucionar esto se puede emplear la publicación de eventos como explicaremos más adelante.

Principio de Sustitución de Liskov (LSP)

Este principio establece que las subclases deben poder sustituir a sus clases base sin cambiar el comportamiento esperado del programa. Para lograr esto las subclases deben garantizar que cumplen el contrato especificado.

Ejemplo: el caso de implementaciones de acceso a datos que utilizan ORMs e implementan el patrón Unit of Work, no persisten los datos en la fuente de datos si no que trabajan con una caché en memoria, esto podría no respetar el contrato especificado y por lo tanto incumpliría el LSP.

Principio de Segregación de Interfaces (ISP)

El ISP establece que las interfaces deben ser pequeñas y específicas basándose en las necesidades del cliente que las utiliza y no en las implementaciones que pudiésemos tener.

Esto evita que las clases implementen métodos que no necesitan, reduciendo así el acoplamiento no deseado y aumentando la cohesión.

Ejemplo: siguiendo el caso anterior, sería una mala definición de una interfaz de acceso a datos definir un método flush para persistir las entidades cacheadas ya que este método puede no ser necesario en otras implementaciones y por lo tanto tampoco es necesario para el cliente que use esa interfaz.

Principio de Inversión de Dependencias (DIP)

El DIP propone que los módulos de alto nivel no deberían depender de los de bajo nivel, ambos deberían depender de abstracciones. Para lograrlo se deben definir interfaces (contratos) de las que depender en vez de las implementaciones, además es importante seguir el principio LSP en este punto.

Este principio permite una mayor flexibilidad, mejora la gestión de dependencias y facilita la realización de tests.

Ejemplo: en los ejemplos de los anteriores principios podemos ver el uso del DIP tanto en las interfaces de acceso a datos como en el publicador de eventos (figura 5.2).

Figura 5.2: Ejemplo de uso del Principio de Inversión de Dependencias (DIP)

5.2 Microservicios

Con el objetivo de construir un servicio robusto, eficiente y altamente escalable se optará por emplear una arquitectura basada en micro-servicios (figura 5.3), este enfoque tiene una serie de ventajas sobre el clásico monolito, a continuación se detallarán los más importantes de cara a cumplir gran parte de los requisitos del sistema:

- **Desacoplamiento:** cada servicio se desarrolla centrándose en un función específica sin necesidad de conocer o depender directamente de otras partes del sistema. Esto agiliza en gran medida el desarrollo y facilita el mantenimiento, pudiendo tener equipos especializados para cada servicio reduciendo el riesgo de incluir errores en otras áreas durante el desarrollo.
- **Escalabilidad:** los micro-servicios pueden escalarse de forma individual tanto vertical como horizontalmente, facilitando la resistencia a picos de demanda momentáneos, por ejemplo, durante la ingesta de datos, o asignando más recursos a un servicio que requiere más carga que el resto.
- **Tolerancia:** en caso de fallo de uno de los servicios, este no afectará al resto garantizando que el sistema pueda seguir funcionando incluso sin alguno de sus componentes.
- **Flexibilidad:** al desacoplar el sistema en funciones diferenciadas, existe la opción de utilizar distintas tecnologías y lenguajes de programación para cada uno de los servicios, esto facilita tanto el desarrollo como mejora el rendimiento, especialmente en proyectos complejos. Además, facilita la integración con sistemas externos, permitiendo que cada servicio se comunique de manera independiente.

Sin embargo, existe un gran desafío a la hora de implementar una arquitectura de micro-servicios, y este es la comunicación. A la hora de implementar una comunicación eficiente entre servicios existen varios enfoques, todos con sus ventajas y desventajas. A continuación se comentarán algunas de las opciones, y en el siguiente apartado se explicará la opción elegida.

- **Compartir infraestructura** (figura 5.4): en esta opción, los servicios no disponen de una infraestructura independiente, por ejemplo, compartiendo la fuente de datos. Esto reduce ampliamente las ventajas de los micro-servicios generando un cuello de botella y un punto único de fallo, lo cual reduce la tolerancia a fallos y empeora notablemente el rendimiento.
- **Restful API** (figura 5.5) en esta ocasión se dispone de una infraestructura específica para cada servicio y aunque es una implementación sencilla, volvemos a encontrarnos

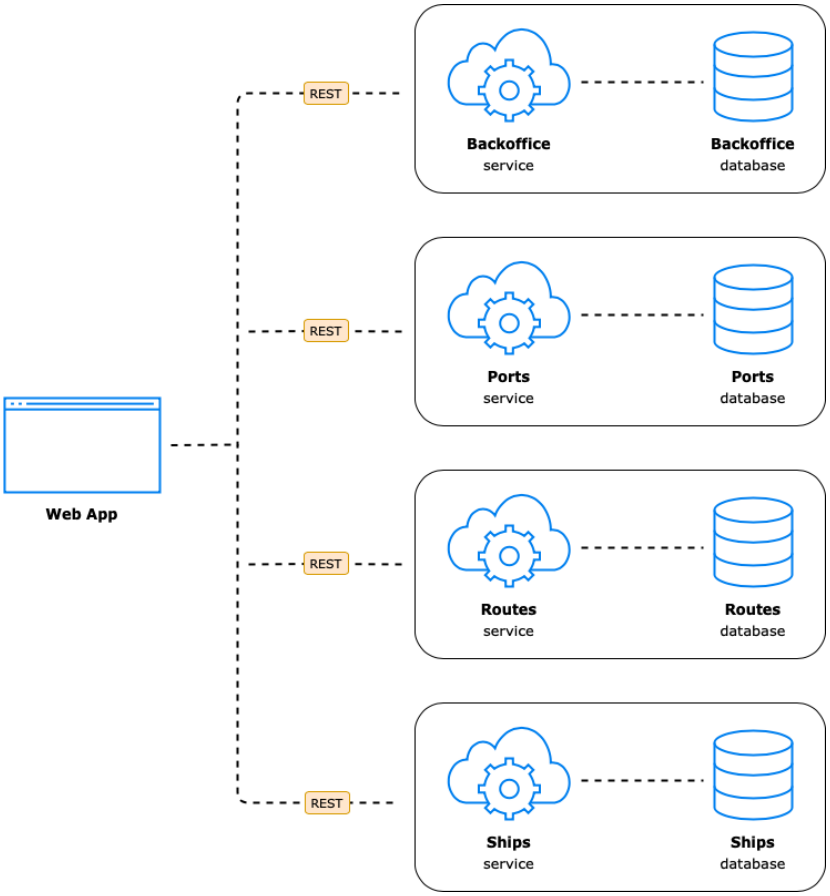


Figura 5.3: Diagrama de arquitectura basada en micro-servicios

con un problema de tolerancia y escalabilidad, ya que los servicios siguen teniendo dependencias directas entre ellos y un fallo en uno de ellos puede provocar la caída del sistema por completo. Además, los protocolos empleados para la comunicación suelen añadir latencia que perjudica el rendimiento.

- **Eventos** (figura 5.6): con este enfoque, los servicios publican eventos a una cola y los servicios que requieran de dicha información pueden suscribirse a ella para almacenar y modelar dicha información para su uso posterior o simplemente ejecutar otra operación que dependa de la misma. A pesar de mantener una gran escalabilidad y tolerancia, para implementar esta opción es necesario añadir complejidad a la infraestructura y ser cuidadoso con la gestión de las colas.

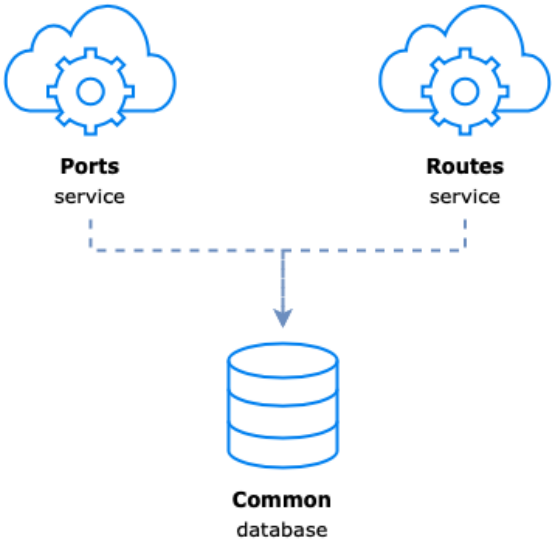


Figura 5.4: Comunicación entre micro-servicios compartiendo infraestructura

5.3 Event-Driven Architecture

Como se explicó en el apartado anterior, una Arquitectura Dirigida a Eventos (EDA) se basa en el intercambio de mensaje o eventos entre los distintos servicios del sistema en lugar de recurrir a llamadas directas entre componentes. Este enfoque nos aporta un desacoplamiento completo entre casos de uso, tanto internos como entre servicios.

La idea es favorecer el principio de responsabilidad única para que cada unos de los casos de uso realice una única operación y publique un evento conforme a que esto ha sucedido. De esta manera otros casos de uso solamente tendrán que suscribirse a ese evento para poder modelar y persistir esa información para su posterior uso o simplemente ejecutar otra operación

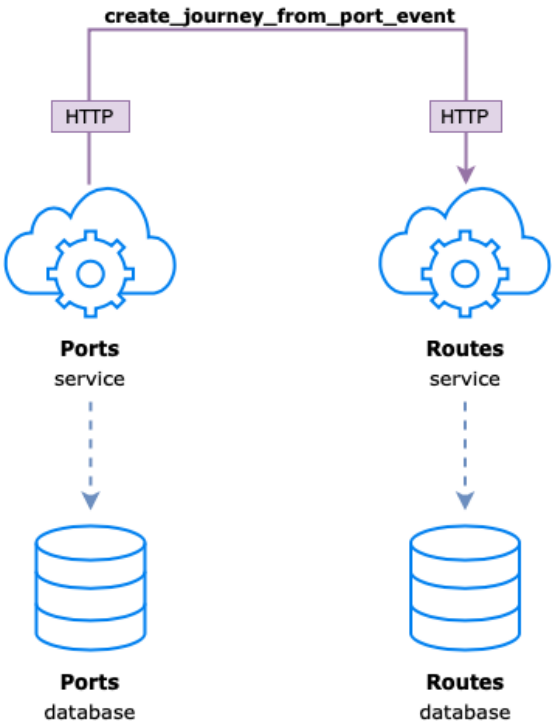


Figura 5.5: Comunicación entre micro-servicios mediante API

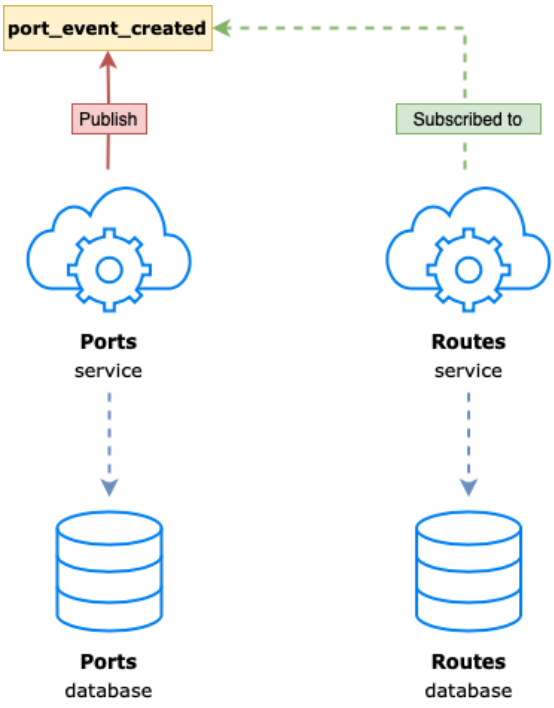


Figura 5.6: Comunicación entre micro-servicios mediante eventos

que depende de la misma. Esto no solo ayuda al desacoplamiento sino que además facilita la labor de desarrollo al agilizar la implementación de nuevas funcionalidades.

5.3.1 Ejemplos

Dos ejemplos claros del uso de eventos serían los siguientes:

- Cuando se crea un trayecto es importante aumentar el número total de trayectos registrados en esa ruta. Con el uso de eventos en vez de aumentar ese contador en el mismo servicio en el que se crea el trayecto, este lanza un evento que escucharía otro servicio encargado de aumentar el contador (figura 5.7).

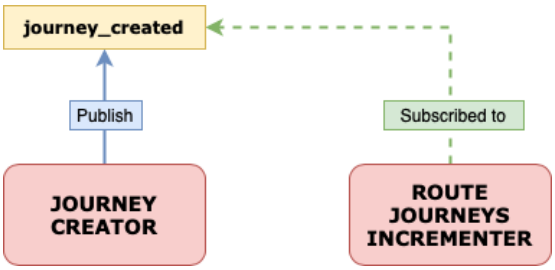


Figura 5.7: Ejemplo de publicación de un evento al crear un trayecto

- La información de los buques registrados es común a varios micro-servicios, para que todos tengan dicha información, el servicio de creación de un buque publica un evento. Este evento es escuchado desde el resto de micro-servicios para persistir la información necesaria de ese buque, y así poder acceder a ella sin depender de otros componentes.

5.3.2 Gestión de errores

A la hora de diseñar un sistema de mensajería nos encontramos con dos potenciales problemas que es necesario atajar, los eventos desordenados y duplicados.

Eventos desordenados

Por la propia naturaleza de las colas de mensajería distribuida no es posible garantizar que los eventos enviados al consumidor estén ordenados. Si bien existen varias soluciones para reencolar los eventos cuando se falla en su consumo evitando así los errores generados por la desordenación, este problema no afecta de forma negativa a nuestro sistema ya que el orden no es crítico en el procesado de los datos.

Eventos duplicados

Este problema es derivado del uso de sistemas distribuidos que priman el rendimiento, si bien existen soluciones como el registro de eventos procesados y la idempotencia, estos no se adecúan al diseño del sistema. La opción elegida para solventar este problema es aplicar una política de reintentos al la hora de procesar un evento cuanto este falla, una vez llegado al máximo de reintentos este eventos se movería a una cola llamada dead letter a la espera de que manualmente se vuelvan consumir.

5.3.3 Otras ventajas

A parte de las ventajas inherentes al uso de un sistema de micro-servicios, un enfoque de comunicación a través de eventos ofrece otras ventajas importantes:

- **Trazabilidad**: los eventos registran todas las acciones y cambios del sistema, lo que facilita la auditoría y trazabilidad.
- **Evolución**: los eventos proporcionan un método muy sencillo para introducir nuevas características al sistema sin necesidad de afectar a las ya existentes simplemente subscribiéndo los nuevos servicios a los eventos relevantes.
- **Integración**: facilita dar acceso a servicios de terceros para integrarse con el sistema sin necesidad de introducir cambios en el mismo, esto ayuda enormemente al desacoplamiento.

5.4 Arquitectura Hexagonal

Una vez vista la arquitectura de hardware del sistema, es esencial definir una arquitectura de software sólida que permita el desarrollo de una aplicación de alta calidad. Se ha optado por seguir el enfoque arquitectónico presentado por Robert C. Martin en su libro ”Clean Architecture”[5][6]. El objetivo principal de esta arquitectura es lograr un código con pocas dependencias y fácil de mantener, promoviendo una serie de principios clave:

- **Independencia de los frameworks:** La arquitectura debe ser independiente de cualquier framework externo, tecnología o biblioteca específica.
- **Independencia de la UI (Interfaz de Usuario):** La lógica de negocio no debe depender de la interfaz de usuario.
- **Independencia de entidades externas:** Las entidades externas, como bases de datos o servicios de terceros, deben ser independientes de la lógica del negocio.

- **Testabilidad:** Las diferentes partes del sistema deben poder probarse de manera aislada para garantizar su correcto funcionamiento.
- **Mantenibilidad:** Debe permitir realizar cambios, mejoras y correcciones de manera eficiente y con un impacto mínimo en otras partes del sistema.

En este proyecto, se ha elegido implementar la Arquitectura Hexagonal (figura 5.8), también conocida como ”Puertos y Adaptadores”. Este patrón arquitectónico es similar a la Arquitectura en Capas y se basa en el Patrón de Inversión de Dependencias y en la separación de responsabilidades.

5.4.1 Capas

1. **Dominio:** Esta es la capa más interna y contiene la lógica de negocio principal. Aquí se definen los conceptos clave del dominio, como Buques, Puertos, Rutas, etc., junto con las interfaces necesarias.
2. **Aplicación:** Es la capa intermedia y en ella residen los casos de uso de nuestra aplicación.
3. **Infraestructura:** La capa más externa contiene todo el código que puede variar según las decisiones externas. Aquí se incluyen las implementaciones concretas de las interfaces definidas en la capa de Dominio, así como la interacción con componentes externos como bases de datos.

5.4.2 Regla de Dependencia

Estas capas definidas anteriormente se relacionan entre sí siguiendo una regla de dependencia que es clave para la correcta implementación de una Arquitectura Hexagonal.

Esta regla dice que el código de cada una de las capas solo deberá conocer los componentes que se ubican en la capa inmediatamente siguiente. Entendiendo el orden de las capas desde fuera hacia dentro: Infraestructura -> Aplicación -> Dominio.

Esta regla lo que nos proporciona es la posibilidad de cambiar elementos de las capas más externas sin que las internas se vean afectadas. Por esto, las piezas que no dependen de nosotros se encuentran en la capa más externa (infraestructura), ya que son las que más posibilidades de variar tienen.

5.4.3 Puertos y Adaptadores

Como bien hemos comentado anteriormente, a Arquitectura Hexagonal también se denomina Puertos y Adaptadores. De hecho, es posible que este término sea más acertado ya

que su nombre no implica limitaciones numéricas. En ese sentido, los puertos y adaptadores hacen referencia a lo siguiente:

- **Puertos:** son las interfaces definidas en la capa de dominio para desacoplarnos de la infraestructura.
- **Adaptadores:** son las implementaciones posibles de esos puertos.

5.4.4 Pruebas

Una de las ventajas clave de la Arquitectura Hexagonal es la facilidad de realizar pruebas. La capacidad de inyectar implementaciones personalizadas para las dependencias de infraestructura permite probar las partes de la aplicación de manera aislada. Esto significa que podemos centrarnos en probar los casos de uso y la lógica del dominio sin preocuparnos por la infraestructura.

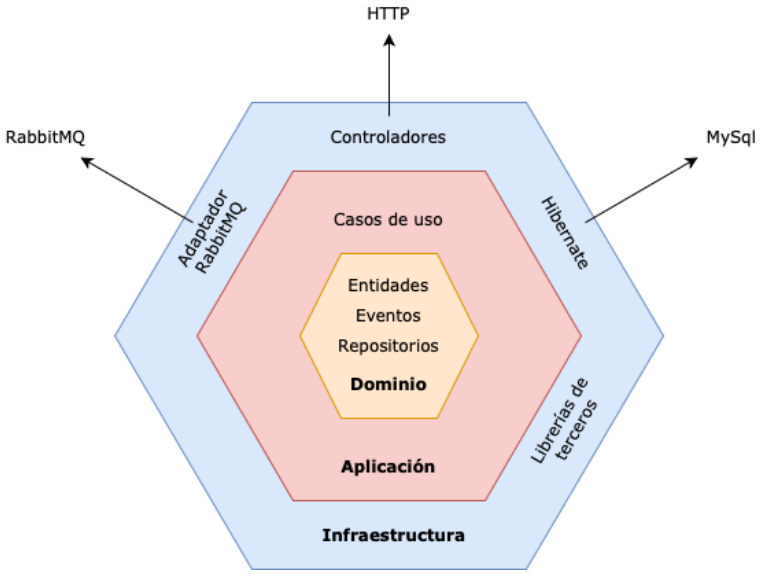


Figura 5.8: Diagrama de Arquitectura Hexagonal

5.5 CQRS

CQRS (Command Query Responsibility Segregation)[7] es un patrón de diseño arquitectónico que separa la lógica de lectura de la lógica de escritura en una aplicación (figura 5.9). Esto es una excelente opción cuando se necesita alta escalabilidad o flexibilidad en la evolución de la aplicación.

5.5.1 Command

Un *Command* representa la intención de realizar una operación en el sistema que acabe modificando el estado de este.

En la práctica, no deja de ser un DTO (objeto de transferencia de datos) que representa la acción que se quiere realizar. Posteriormente, este *Command* llegaría a un *CommandHandler* que es el encargado de realizar las validaciones básicas de integridad de los datos y delegar en el caso de uso para realizar la acción.

5.5.2 Query

Por el contrario, una Query representa la intención de pedir ciertos datos al sistema sin que ello acabe alterando el estado del mismo. Esto último facilita cachear las respuestas de estas operaciones.

Al igual que vimos anteriormente, una *Query* simplemente es un DTO que representa la petición que queremos consultar. Al igual que con los *Commands*, también existe un *QueryHandler* que validaría la integridad de los datos y delegaría en el caso de uso para devolver una *Response*.

5.5.3 Command y Query Bus

El *Bus* es el medio por el cual los *Commands* y las *Queries* llegan a sus respectivos *Handlers*, y aquí es donde residen varios beneficios de CQRS, ya que al tener dos buses separados para la lectura y la escritura se pueden ajustar a necesidades distintas.

Un ejemplo sería el uso de un *Command Bus* asíncrono cuando las operaciones de escritura son muy costosas para no bloquear al cliente durante un tiempo excesivo.

Otra posibilidad es la comentada anteriormente de cachear respuestas a una *Query* desde el propio *Query Bus* para agilizar los tiempos de consulta del cliente.

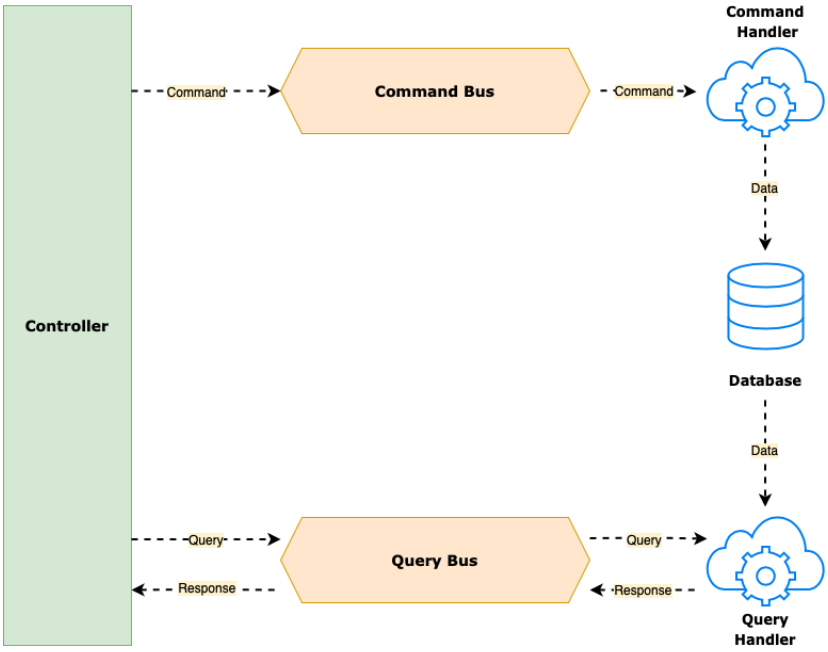


Figura 5.9: Diagrama de CQRS

Capítulo 6

Desarrollo

En este capítulo se ahondará en la implementación del sistema, haciendo hincapié en los puntos claves de todo el proceso de desarrollo.

6.1 Tecnologías

Antes de comenzar con los detalles de la implementación es importante tener constancia de las tecnologías necesaria para llevar a cabo el proyecto. A continuación, se exponen las más importantes.

6.1.1 Lenguajes

- **Java 11**: El lenguaje principal utilizado en el proyecto para la implementación de la lógica de negocio y el backend de la aplicación. Java proporciona una plataforma sólida y madura ideal para desarrollar cualquier tipo de aplicación.
- **Groovy**: Groovy se empleó junto con Gradle para la gestión de dependencias y la automatización de tareas en el backend.
- **SQL**: Se utilizó SQL para interactuar con la base de datos y definir las estructuras de datos del backend.
- **HQL**: Hibernate Query Language (HQL) se usó para consultar y actualizar objetos en la base de datos utilizando consultas orientadas a objetos previamente definidas.
- **YAML**: Se empleó YAML en la configuración de Docker Compose para poder ejecutar todos los servicios e infraestructura en local.
- **HTML**: El lenguaje de marcado que se empleó para definir la estructura de las vistas de la interfaz de usuario.

- **CSS**: Es un lenguaje de estilado utilizado para definir el estilo visual de los elementos de la interfaz de usuario.
- **JavaScript**: Lenguaje de programación empleado para dotar de funcionalidad e interactividad a la interfaz de usuario.

6.1.2 Librerías

- **Spring**: El framework Spring se utilizó para desarrollar el backend de la aplicación. Spring nos proporciona una gran cantidad de módulos que facilitan el desarrollo, en este caso se emplearon:
  - **Spring Boot Web**: proporciona las herramientas necesarias para desarrollar aplicaciones web de manera eficiente.
  - **Spring Boot Security**: ofrece una serie de utilidades para implementar la autorización en la aplicación.
  - **Spring Boot Batch**: utilizada para procesar los datos de ingesta por lotes de manera eficiente.
  - **Spring Boot Amqp**[8]: facilita la integración con el sistema de mensajería escogido, RabbitMQ.
  - **Spring Boot Test**: proporciona las herramientas necesarias para implementar correctamente las pruebas unitarias y de integración de la aplicación.
  - **Spring ORM**: facilita la interacción con la base de datos a través de Hibernate simplificando el acceso y actualización.
- **Hibernate**: Se empleó para mapear entidades en Java a objetos de base de datos y gestionar las consultas y actualizaciones de manera sencilla.
- **Reflections**[9]: Es una biblioteca que facilita la búsqueda de clases mediante reflexión, es empleada para la implementación de los diferentes buses.
- **Searoute**[10]: Librería desarrollada por Eurostat que facilita el cálculo de la ruta marítima mas corta entre dos localizaciones.
- **JUnit**: El framework principal para realizar las pruebas unitarias en el backend de la aplicación.
- **Mockito**: Librería utilizada para simular el comportamiento de ciertos componentes del sistema con el fin de no tener que depender de la infraestructura al realizar las pruebas unitarias.

- **Faker**: Librería empleada para generar datos de prueba durante la ejecución de las pruebas.
- **React**: Librería de JavaScript utilizada para el desarrollo de interfaces de usuario siguiendo un paradigma basado en componentes.
- **Mapbox**[11]: Librería utilizada para la visualización de mapas e información geográfica en la interfaz de usuario.
- **Material-UI**[12]: Biblioteca de componentes para React empleada para mantener un diseño homogéneo y facilitar la implementación de estilos en la interfaz de usuario.

6.1.3 Infraestructura

- **MySQL**: Elegido como el sistema de gestión de bases de datos para almacenar la gran cantidad de datos que maneja la aplicación.
- **RabbitMQ**: Utilizado como el sistema de mensajería para implementar la comunicación entre los distintos servicios del sistema de forma asíncrona.
- **Docker**: Empleado para contenerizar los distintos servicios facilitando el desarrollo y posterior despliegue en producción.

6.1.4 Herramientas

- **IntelliJ IDEA**: Entorno de desarrollo integrado (IDE) empleado tanto para la codificación y pruebas como para acceder y gestionar las bases de datos.
- **Gradle**: Empleado para gestionar dependencias y ejecutar distintas tareas del ciclo de vida de desarrollo de los servicios en el backend.
- **Make**: Empleado para facilitar la ejecución de diferentes tareas en el backend de la aplicación como son la ejecución de tests o la contenerización de los micro-servicios.
- **NPM**: Utilizado para gestionar las dependencias y paquetes utilizados en el frontend.
- **Git - GitHub**: Se usó Git como sistema de control de versiones distribuido, y GitHub como repositorio en la nube y para ejecutar acciones de integración y despliegue.
- **Postman**: Aplicación empleada para probar las API de la aplicación, lo que facilitó la integración entre backend y frontend.
- **DigitalOcean**: [13] Plataforma de alojamiento en la nube utilizada para disponibilizar la aplicación en un entorno de producción.

6.2 Estructura

El código de la aplicación se estructura en dos bloques separados, backend y frontend, cada uno con sus peculiaridades.

6.2.1 Backend

Como se comentó con anterioridad, para desarrollar el backend de la aplicación se optó por utilizar una arquitectura enfocada en microservicios. Para reducir un poco el aumento de complejidad se decidió implementar todos estos en un mono-repositorio con una estructura un tanto particular.

El proyecto es gestionado por Gradle utilizando sub-proyectos configurados de manera personalizada para favorecer la implementación.

Dentro del directorio */src* es donde nos encontramos los diferentes sub-proyectos con toda la lógica y casos de uso de la aplicación, haciendo referencia a los diferentes microservicios (figura 6.1a):

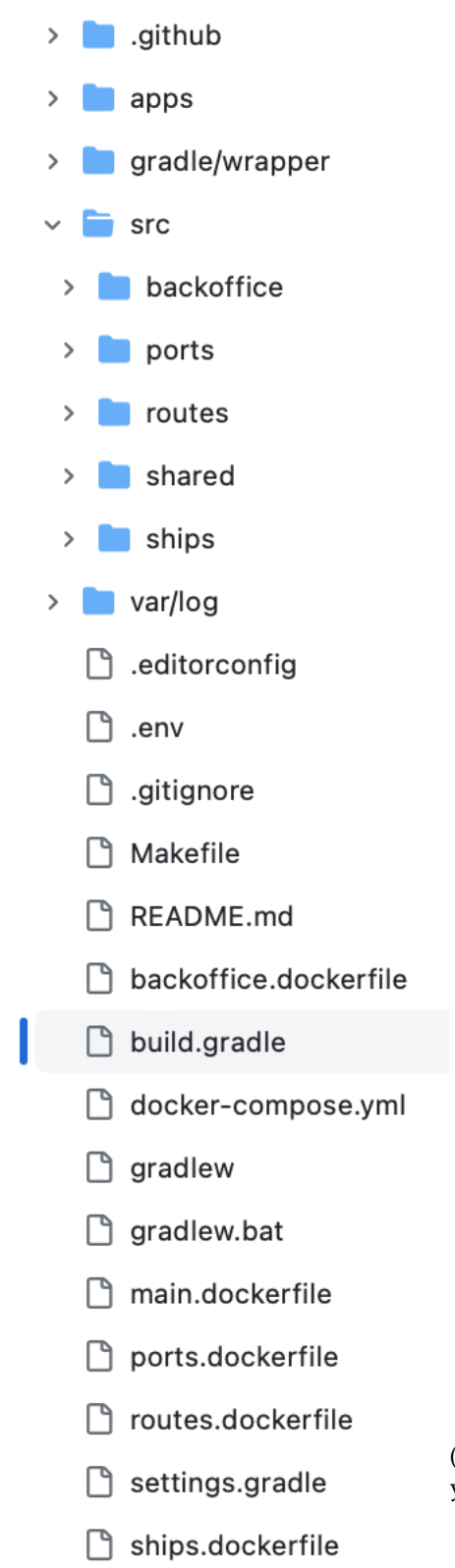
- */src/backoffice*: lógica y casos de uso para realizar las ingestas de datos y gestionar usuarios.
- */src/ports*: lógica y casos de uso relacionados con puertos y eventos portuarios.
- */src/routes*: lógica y casos de uso relacionados con rutas y trayectos marítimos.
- */src/ships*: lógica y casos de uso relacionados con buques.

A mayores de esto cuatro, nos encontramos con un sub-proyecto */shared* donde se alojará todas las entidades, componentes y utilidades comunes a los distintos micro-servicios como pueden ser los buses de eventos o las implementaciones para conectarse con la infraestructura.

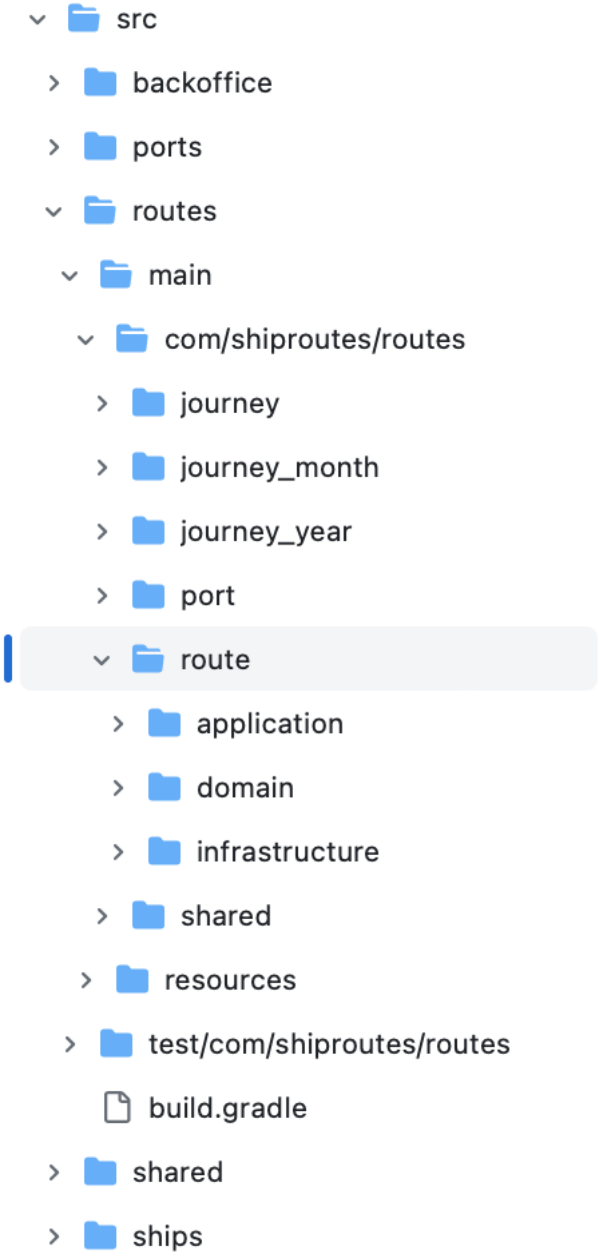
El proyecto principal, cuyo código se encuentra en el directorio */apps*, es el que hace la función de entrada a la aplicación y en él se encuentran todos los controladores de la API Rest así como las herramientas necesarias para desarrollar comandos para realizar tareas específicas. Dentro de este directorio también nos encontramos la misma división de micro-servicios comentada anteriormente, teniendo cada uno sus puntos de entrada específicos.

Por supuesto, cada uno de los proyectos se divide a su vez en los directorios */main* donde nos encontraremos el código de producción y el directorio */test* donde se encontrarán todas las pruebas automáticas de dicho proyecto.

A su vez los sub-proyectos (figura 6.1b) se dividen internamente en diferentes módulos relacionados con una entidad concreta (puerto, evento portuario, ruta...), en donde encontramos una estructura que sigue los estándares de la Arquitectura Hexagonal con tres directorios separados para infraestructura, aplicación y dominio.



(a) Estructura general del backend



(b) Estructura de un subproyecto del backend

Figura 6.1: Estructura del backend

6.2.2 Frontend

Por su parte, la estructura del frontend (figura 6.2)es es bastante más simple, ya que es la estructura básica de un proyecto React. Nos encontraremos con un directorio */node_modules* donde se cargarán las dependencias de la aplicación, el directorio */public* donde se encuentran todos los ficheros estáticos como imágenes que utiliza la aplicación y el directorio */src* en donde se aloja todo el código que compone la interfaz de usuario.

El directorio */src* se organiza a su vez en diferentes directorios que agrupan componentes de temática similar y un directorio */services* donde ser realizan las peticiones HTTP para comunicarse con el backend.

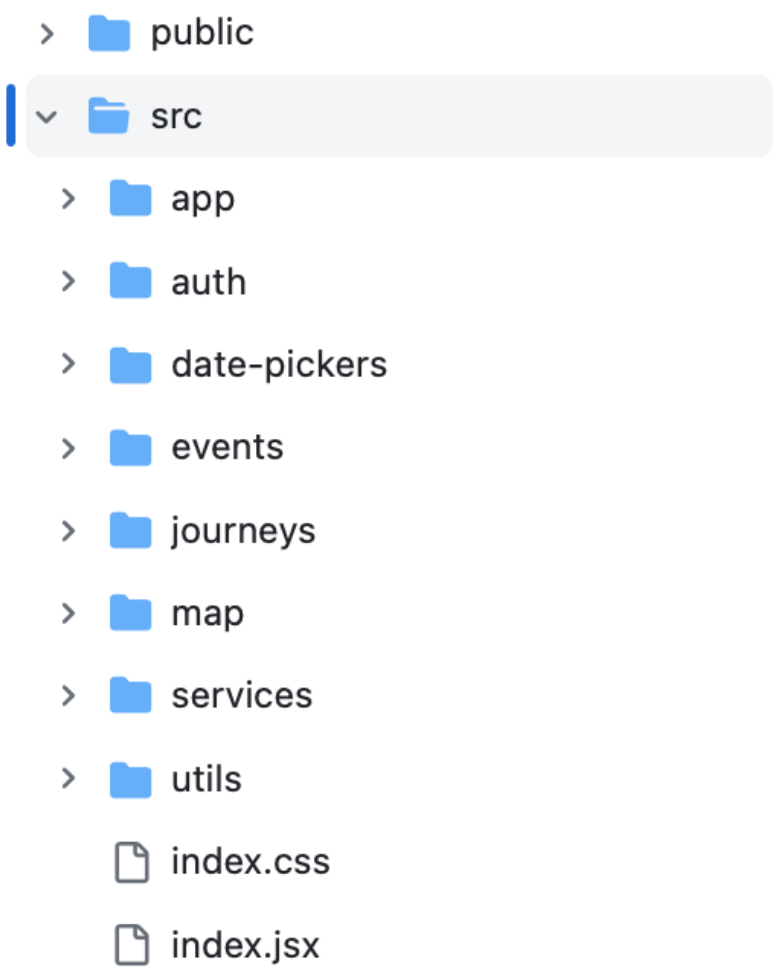


Figura 6.2: Estructura general del frontend

6.3 Modelo de datos

Las entidades principales del sistema son *Buque* y *Puerto*, combinando estas dos nos encontramos con las siguientes entidades:

- *Evento Portuario*: salida o llegada de un buque a un puerto en una fecha determinada.
- *Ruta*: unión entre dos puertos, uno de salida y otro de llegada.
- *Trayecto*: desplazamiento de un barco por una ruta con fecha de salida y llegada.

A mayores también existe la entidad *Usuario* empleada para la identificación y autorización. Puede verse el diagrama de base de datos en la figura 6.3

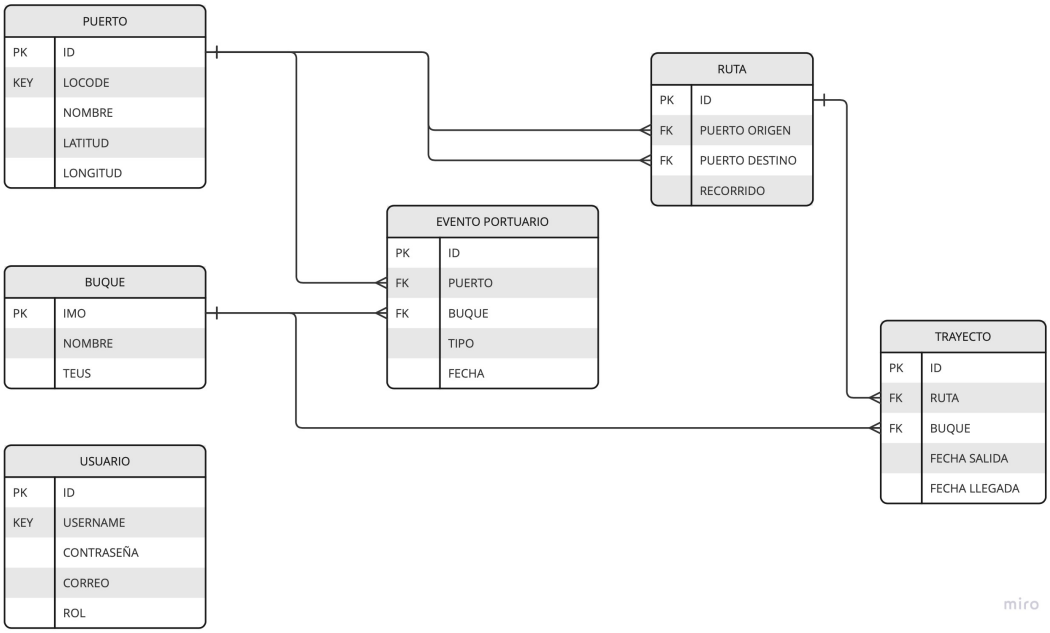


Figura 6.3: Diagrama de base de datos normalizado

6.4 Optimización

Con las seis entidades anteriormente mencionadas, se podrían satisfacer todos los casos de usos necesarios actualmente en la aplicación, pero a la hora de analizar su rendimiento nos encontramos con bastantes deficiencias debido a la gran cantidad de datos con la que es necesario trabajar en las consultas.

Para solucionar este problema se han empleado dos técnicas de optimización de consultas que, si bien reducen el rendimiento de la inserción y actualización al disponer de datos

repetidos, sustentadas en la comunicación a través de colas de mensajes y la publicación de eventos de dominio aumentan considerablemente el rendimiento de la aplicación.

6.4.1 Desnormalización

Con el fin de evitar consultas SQL con *joins* muy costosos se optó por incluir en cada una de las entidades todos los campos necesarios para completarla (figura 6.4), por ejemplo en la entidad *Evento Portuario* en vez de definir únicamente el identificador del *Puerto*, también se añaden su nombre y coordenadas.

Esto nos ayuda enormemente a aligerar las consultas a base de datos al necesitar solamente acceder a una de las tablas y no tener que unir cada una de las cientos de miles de instancias con su respectiva instancia de la entidad *Puerto*, *Buque*, etc.

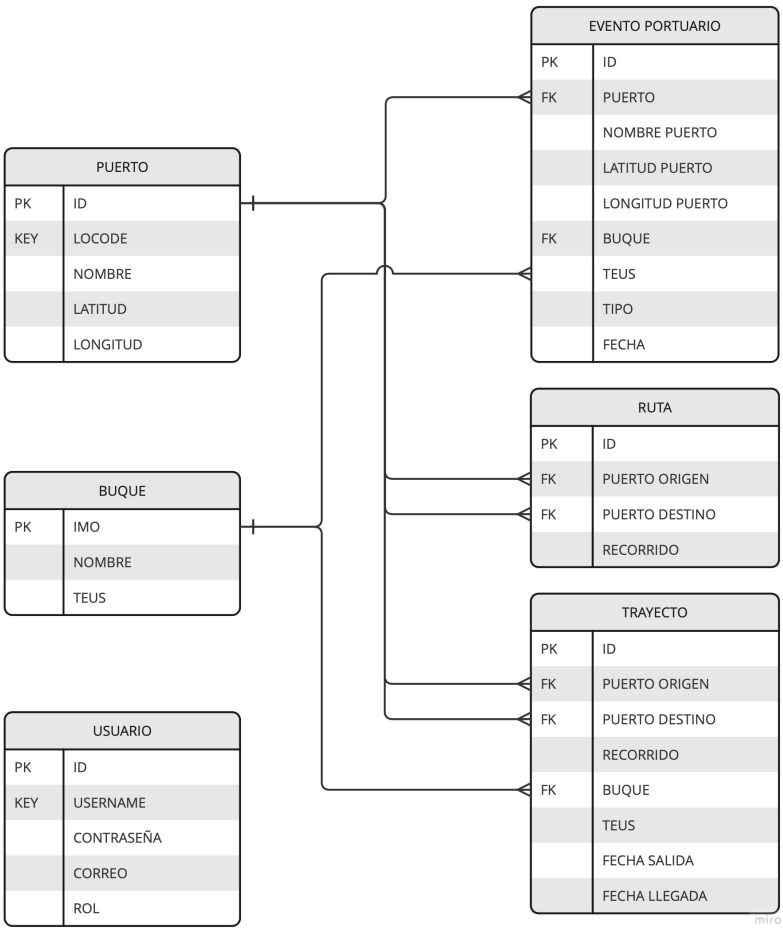


Figura 6.4: Diagrama de base de datos desnormalizado

6.4.2 Agregación

Uno de los requisitos de la aplicación es poder obtener datos agregados por años y meses tanto de la entidad *Evento Portuario* como de *Trayecto*, ambas contienen millones de instancias lo que hace las consultas con agregaciones muy costosas.

Para solucionar este problema se añaden cuatro nuevas entidades: *Eventos Portuarios Anuales*, *Eventos Portuarios Mensuales*, *Trayectos Anuales* y *Trayectos Mensuales*. Estas entidades (figura 6.5) servirán para acceder a esos datos agregados mejorando considerablemente el rendimiento de las consultas y serán actualizadas a través de eventos de dominio al crear, modificar o eliminar instancias de la entidad base.

6.5 Implementación del backend

Para la implementación del backend se ha tenido muy en cuenta todo lo definido en el apartado de diseño para crear una aplicación acorde a los estándares y principios definidos.

Antes de comenzar a comentar los puntos claves de la implementación es importante apuntar que lo primero que se tuvo en cuenta a la hora de implementar cada uno de los casos de uso y modelar cada una de las entidades fue eliminar por completo las dependencias directas con terceros en las capas de dominio y aplicación, incluyendo tanto librerías externas como dependencias con el propio framework.

6.5.1 Entidades

Las entidades han sido desarrolladas utilizando *Value Objects*[14] para encapsular conceptos de nuestro sistema que se identifican por su contenido, un ejemplo de esto puede ser el identificador de un buque (figura 6.6) o las coordenadas de un puerto. Su uso aporta los siguientes beneficios:

- Encapsula la lógica y la empuja hacia nuestro dominio.
- Unifica las validación de integridad de los datos.
- Aporta semántica a los métodos.

Las entidades también son las encargadas de crear y registrar los eventos de dominio relacionados sobre la misma, de este modo cuando una de nuestras entidades se crea o modifica, es la propia entidad la que crea y almacena el evento correspondiente a dicha operación (figura 6.7).

Para conseguir esto último es muy importante que las entidades no sean simples almacenes de datos, si no que hay que mover toda la lógica referida a la misma dentro de propia entidad.

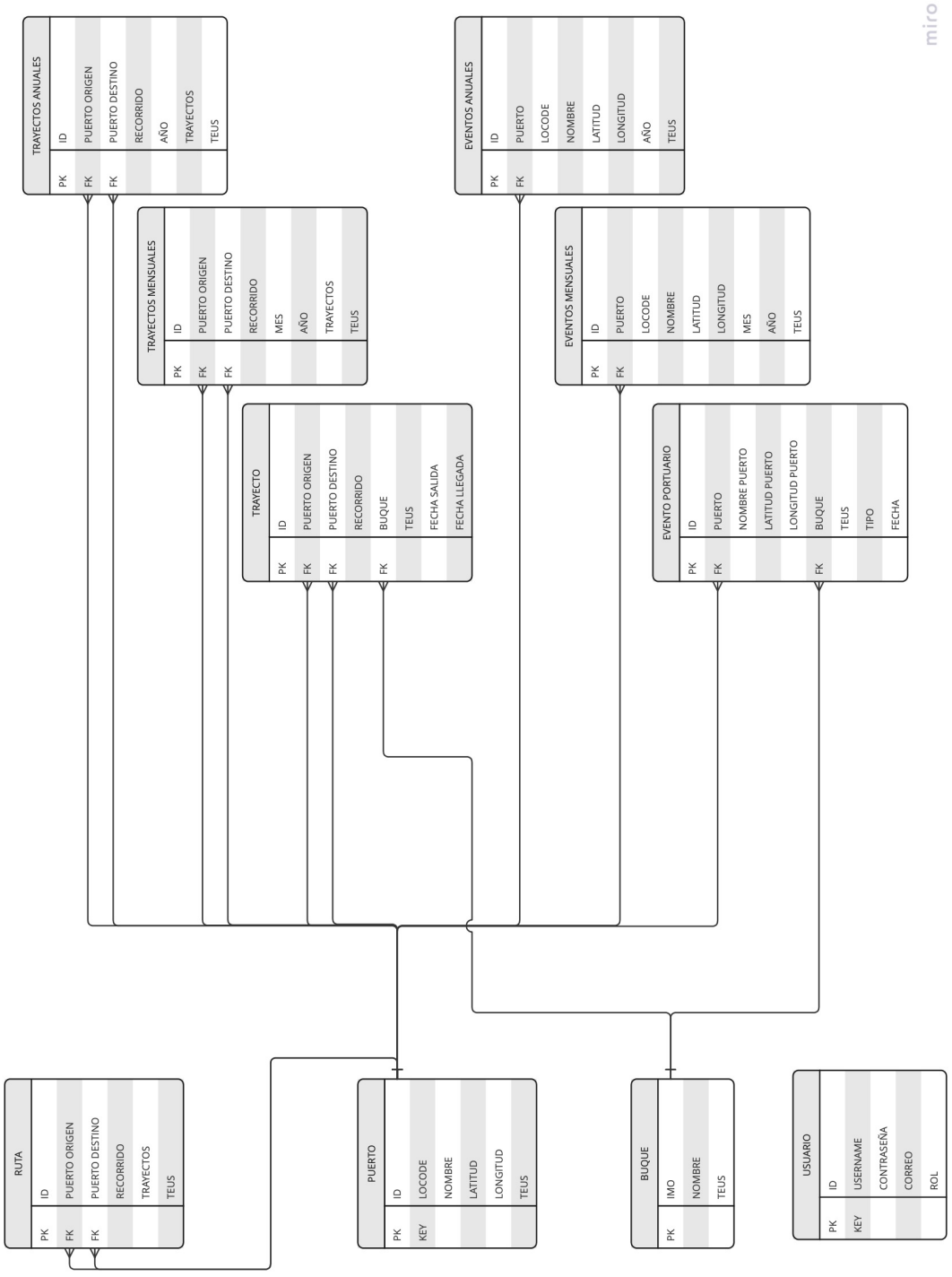


Figura 6.5: Diagrama de base de datos final

```java
public final class IMO implements Serializable {
    5 usages
    private final String value;

    Diego Seoane
    public IMO(String value) throws IllegalArgumentException {
        ensureValidIMO(value);
        this.value = value;
    }

    1 usage   Diego Seoane
    private void ensureValidIMO(String value) throws IllegalArgumentException {
        if (value.length() ≠ 7) {
            throw new IllegalArgumentException("Invalid IMO length: " + value + ". IMO must be a 7-digit number");
        }
        Integer[] digits;
        try {
            digits = Arrays.stream(value.split( regex: "")).map(Integer::parseInt).toArray(Integer[]::new);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Invalid IMO: " + value + ". IMO must be a 7-digit number");
        }
        int checkDigit = digits[6];
        int product = 7 * digits[0] + 6 * digits[1] + 5 * digits[2] + 4 * digits[3] + 3 * digits[4] + 2 * digits[5];
        if (checkDigit ≠ product % 10) {
            throw new IllegalArgumentException("Invalid IMO: " + value);
        }
    }

    new *
    public String value() {
        return value;
    }
```

Figura 6.6: Value Object del indentificador de un buque (IMO)

```java
public static Ship create(IMO imo, ShipName name, Teus teus) {
    Ship ship = new Ship(imo, name, teus);

    ship.record(new ShipCreatedEvent(imo.value(), name.value(), teus.value()));

    return ship;
}
```

Figura 6.7: Registro de un evento en la propia entidad

6.5.2 Entidad de base de datos

Cada una de las entidades de nuestro dominio tiene asociada una entidad en infraestructura encargada de definir el modelado de esos datos en la propia base de datos. De esta manera se eliminan las dependencias con la infraestructura y con el propio framework de nuestro dominio.

Esta entidad de base de datos tiene completo conocimiento de la entidad de dominio y es la encargada de implementar la conversión de una a otra (figura 6.8).

```java
@Entity(name = "ship")
@Table(name = "ships")
public final class HibernateShipEntity {

    @Id
    private String imo;

    2 usages
    private String name;

    2 usages
    private Integer teus;

    Diego Seoane
    public HibernateShipEntity() {
    }

    1 usage  Diego Seoane
    public HibernateShipEntity(Ship ship) {
        this.imo = ship.imo().value();
        this.name = ship.name().value();
        this.teus = ship.teus().value();
    }

    1 usage  Diego Seoane
    public Ship toEntity() {
        return new Ship(
            new IMO(this.imo),
            new ShipName(this.name),
            new Teus(this.teus)
        );
    }

}
```

Figura 6.8: Entidad de base de datos

6.5.3 Creación de trayectos

La mayor parte de los casos de uso tienen una implementación bastante trivial, sobretodo gracias al uso de eventos de dominio que simplifica las operaciones dependientes de la prin-

cipal. Pero existe un caso de uso clave que implica cierta complejidad en la implementación, este es el de creación de trayectos a partir de eventos portuarios (figura 6.9).

Lo primero es definir cuándo dos eventos portuarios forman un trayecto, para que esto suceda se deben cumplir las siguientes condiciones:

- Deben ser realizados por el mismo buque.
- Uno debe ser una salida y el otro una llegada.
- La salida debe ser posterior a la llegada.
- No debe haber registrado ningún otro evento portuario del mismo buque entre ambos eventos.

Si tuviésemos la certeza de que los eventos portuarios se registran siempre de forma ordenada la implementación sería bastante trivial, pero esto no es factible. Debido a esto podemos encontrarnos que lo que en cierto momento es un trayecto válido, puede dejar de serlo al registrarse un nuevo evento portuario.

Para solucionar este problema se sigue el siguiente procedimiento para poder crear y actualizar correctamente los trayectos:

1. Se recibe un evento portuario y se comprueba si es una salida o una llegada.
2. Se busca el evento portuario complementario registrado más cercano en el tiempo que no tenga asociado otro evento más cercano que el recibido.
3. Si no existe dicho evento portuario complementario, se almacena el evento como un trayecto incompleto.
4. Si el evento portuario complementario existe y forma parte de un trayecto incompleto se actualiza con la información del evento portuario recibido y se almacena.
5. Si el evento portuario complementario existe y ya forma parte de un trayecto completo se hace lo siguiente:
   (a) Se elimina el trayecto.
   (b) Se crea y almacena un nuevo trayecto con el evento portuario recibido y el complementario.
   (c) Se vuelve a realizar el proceso desde el primer paso con el evento portuario que ha quedado desvinculado.

Además, la implementación queda bastante simplificada al publicar un evento de dominio cuando se queda desvinculado un evento portuario. De esta manera la posible cascada de operaciones no se produce en la operación principal y esto la hace independiente y mejora su rendimiento.

```
public void create(IMO shipId, Teus teus, PortId originPort, DepartureDate departureDate) {
    JourneyId journeyId = new JourneyId(uuidGenerator.generate());

    Optional<Journey> optionalJourney = repository.searchJourneyArrival(shipId, departureDate);

    if (optionalJourney.isEmpty()) {
        Journey newJourney = Journey.departure(journeyId, shipId, teus, originPort, departureDate);
        repository.save(newJourney);
        return;
    }

    Journey journeyArrival = optionalJourney.get();

    RoutePath routePath = getPath(originPort, journeyArrival.destinationPort());
    Journey journey = Journey.create(journeyId, shipId, teus, originPort, journeyArrival.destinationPort(),
        departureDate, journeyArrival.arrivalDate(), routePath);

    repository.remove(journeyArrival);
    repository.save(journey);
    if (journeyArrival.isComplete()) {
        journeyArrival.recordRemovedEvent();
        journeyArrival.recordUnlinkedDepartureEvent();
    }
    eventBus.publish(journey.pullDomainEvents());
    eventBus.publish(journeyArrival.pullDomainEvents());
}
```

Figura 6.9: Algoritmo para la creación de un trayecto

6.5.4 Generación de recorridos

Para calcular el recorrido aproximado que seguiría un buque para cubrir una ruta se emplea una librería creada por Eurostat llamada Searoute[10], la cual permite calcular la ruta marítima más corta entre dos localizaciones.

Si bien la librería satisface las expectativas en la amplia mayoría de las ocasiones, hay ciertos casos en los que desordena las coordenadas seguidas por la ruta al cruzar los meridianos 0º y 180º. Por esta razón fue necesario añadir código adicional para validar y solucionar el resultado generado por la librería.

6.5.5 Ingesta de datos

La ingesta masiva de datos se implementó utilizando Spring Batch[15] para procesar las entradas del CSV por lotes optimizando así la validación e ingesta de los mismos.

El procedimiento es sencillo, en cada entrada nos encontramos con la información básica necesaria del puerto, el buque y el evento portuario, y se siguen los siguientes pasos:

1. Validación de los datos.
2. Creación del puerto si todavía no está registrado.
3. Creación del buque si todavía no está registrado.
4. Registro del evento portuario.

Para no ralentizar el procesado de las entradas, la creación de las nuevas entidades se realiza emitiendo un nuevo evento de dominio para que así todo el procedimiento se realice de forma asíncrona.

6.6 Implementación del frontend

Debido a la poca experiencia desarrollando interfaces de usuario, para la implementación de esta se optó por simplificar el desarrollo al máximo tratando de emplear herramientas ya existentes y creando una interfaz y un diseño sencillo.

6.6.1 Mapas

Para la implementación de los mapas se empleó MapBox[11], una librería muy completa que permite crear mapas interactivos con distintas capas y personalizar tanto su diseño como comportamiento de manera sencilla.

Para la vista de puertos empleamos una funcionalidad de MapBox que nos permite agrupar datos en función de su proximidad geográfica creando "clusters"[16]. Esta función junto con varios ajustes de funcionamiento y estilo nos permitió crear fácilmente una solución para la visión agrupada por zonas de los eventos portuarios.

Sin embargo, en la visualización de rutas no encontramos un pequeño problema al representar líneas de puntos que pasaban por el meridiano de 180º, esto es debido a una limitación de MapBox que solo permite dibujar líneas sobre el mapa en un único sentido. Para solucionar esto fue necesario procesar los recorridos de las rutas para dividirlas en dos líneas separadas en caso de que estas pasasen por dicho meridiano.

6.6.2 Estado

La gestión del estado de la aplicación se realiza en su mayoría en cada uno de los componentes de la aplicación debido a que no se encontró necesario gestionar de forma global el estado.

La única excepción que encontramos es el caso de la información de un usuario autenticado, que se gestiona empleando React Context. Esto nos permite poder compartir ciertas partes muy acotadas del estado entre los componentes que requieren su uso en cualquier parte de la aplicación.

6.6.3 Estilo

Para darle estilo a la interfaz se empleó Material-UI[12], una librería de componentes de React que facilita la creación de una interfaz de usuario homogénea y atractiva. Esta librería proporciona una amplia variedad de componentes personalizables y predefinidos que se integran bien en la aplicación.

6.7 Contenerización y despliegue

Para facilitar el despliegue y la ejecución de la aplicación durante el desarollo se optó por contenerizar tanto los servicios del backend como el frontend utilizando Docker. Además para gestionar los distintos servicios del backend en el entorno de desarollo local también se empleó Docker Compose.

Una vez subido el código al repositorio remoto de GitHub, se utiliza la funcionalidad de las GitHub Actions para poder compilar el proyecto y generar una imagen de Docker para cada uno de los servicios. Una vez creada la imagen esta se sube al registro de contenedores de Digital Ocean[13] donde automáticamente se despliega, quedando disponible en cuestión de minutos (accesible en el siguiente enlace https://ship-routes-map-gldqb.ondigitalocean.app.

Capítulo 7

Conclusiones y trabajo futuro

El objetivo final de este proyecto no era otro que conseguir una herramienta que pudiese ser de utilidad para investigadores y analistas de cara a comprender y analizar las distintas redes de transporte marítimo además de facilitar la gestión de esa gran cantidad de datos. Pero el sistema no debería quedarse ahí, debía construirse un proyecto a futuro, y sentar las bases para poder ampliar los usos y utilidades del producto desarrollado.

Tras muchas horas de estudio del contexto, análisis, diseño y desarrollo, se ha podido crear una aplicación intuitiva y usable, que no solamente puede satisfacer las necesidades de expertos sino que también sirve como puerta de entrada a personas con curiosidad por conocer más acerca de la logística marítima. Esto se ha conseguido gracias a una interfaz limpia y sencilla que prima las referencias geográficas en la visualización de los datos (figuras 7.1 y 7.2).

7.1 Trabajo realizado

En lo referente al trabajo que se ha realizado, la complejidad y tamaño del proyecto en líneas generales ha sido muy similar a lo esperado, pero es cierto que ciertas partes del trabajo a realizar han supuesto un mayor esfuerzo del esperado en contraposición de otras que se han superado con mayor facilidad.

El trabajo de análisis y limpieza de los datos iniciales de lo que se disponía ha sido una tarea bastante ardua que tardó en llevarse a cabo bastante más de lo esperado. Lo mismo sucedió con la visualización de las rutas marítimas, ciertas limitaciones en librerías de terceros junto a una lógica más complicada de lo esperado hicieron que estos componentes fundamentales de la aplicación supusiesen mayor esfuerzo el estimado.

Por contraparte, las decisiones de diseño y de estilo en el desarrollo agilizaron más de lo esperado la implementación de los casos de uso, el modelado de las entidades y el diseño de pruebas automáticas.

7.1.1 Lecciones aprendidas

Durante la ejecución del proyecto se han podido adquirir nuevas competencias muy valiosas así como una grandísima experiencia en el uso de tecnologías de gran uso en el sector de la ingeniería de software.

Durante el desarrollo se han podido poner en práctica una gran cantidad de principios y patrones con los cuales nunca se había trabajado, como son la aplicación consciente de los principios SOLID, o la implementación de CQRS en el proyecto. De la misma forma ha servido como formación en la gestión de micro-servicios así como en el uso de comunicación asíncrona, conceptos con los que no se había trabajado anteriormente.

Pero no solo se han obtenido conocimientos puramente técnicos, la realización del proyecto también ha sido una gran oportunidad para aplicar conocimientos previamente adquiridos sobre metodología, gestión de proyectos y organización. Además también ha servido para aprender a realizar estimaciones de mayor calidad.

Todo esto son lecciones y aprendizajes que aportan un gran valor de cara al desarrollo laboral, y que no habrían sido tan fáciles de conseguir si la libertad y flexibilidad de realizar un proyecto propio en el que experimentar y aprender.

7.1.2 Trabajo futuro

Como ya se ha comentado anteriormente, el objetivo es que la aplicación no se quede en lo desarrollado en este proyecto y pueda seguir creciendo en funcionalidad y usos, con el fin de aportar un mayor valor. A continuación, se exponen algunas líneas sobre las que seguir trabajando:

- Dotar de mayor profundidad a los análisis concretos de la red incorporando el uso de filtros más complejos, para poder personalizar las visualizaciones acotando los espacios de búsqueda.
- Valorar la incorporación de nuevas visualizaciones para aportar distintos enfoques como la comparación del estado de la red a lo largo del tiempo, las conexiones comerciales entre países o el centrarse únicamente en un elemento concreto de la red como un puerto o buque.
- Explorar la posibilidad de integrar modelos de aprendizaje automático para predecir patrones, estimar tiempos y detectar anomalías.
- Implementar un sistema de notificación y alertado para ayudar a localizar anomalías.

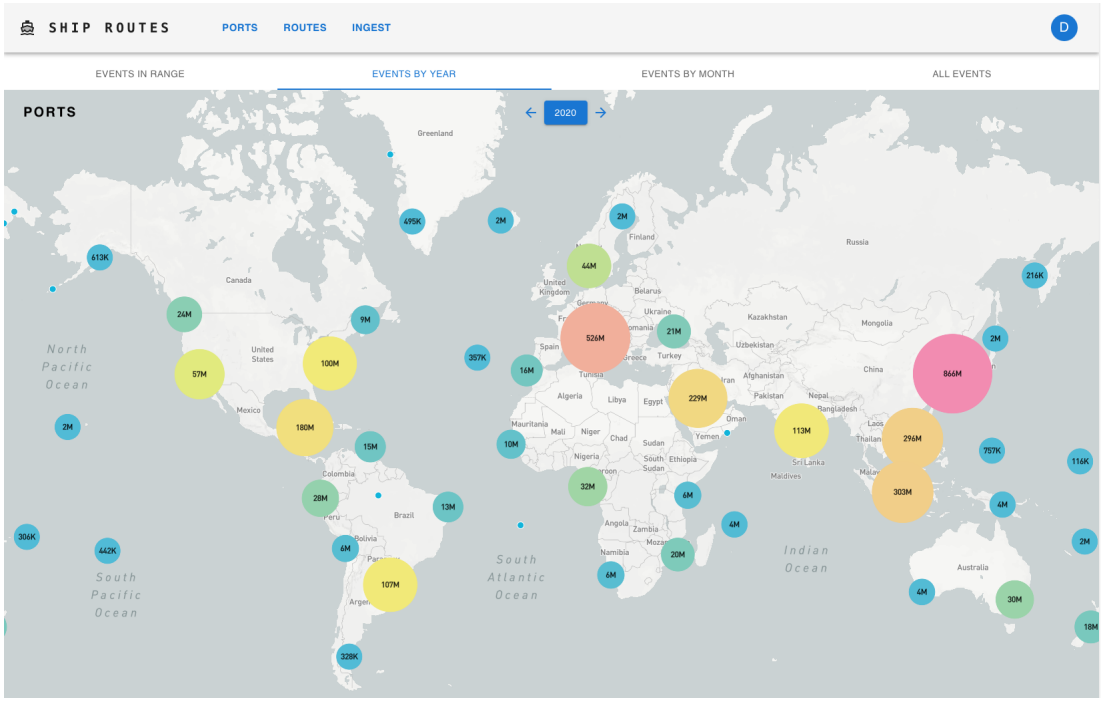


Figura 7.1: Ejemplo de visualización de la situación portuario en 2020

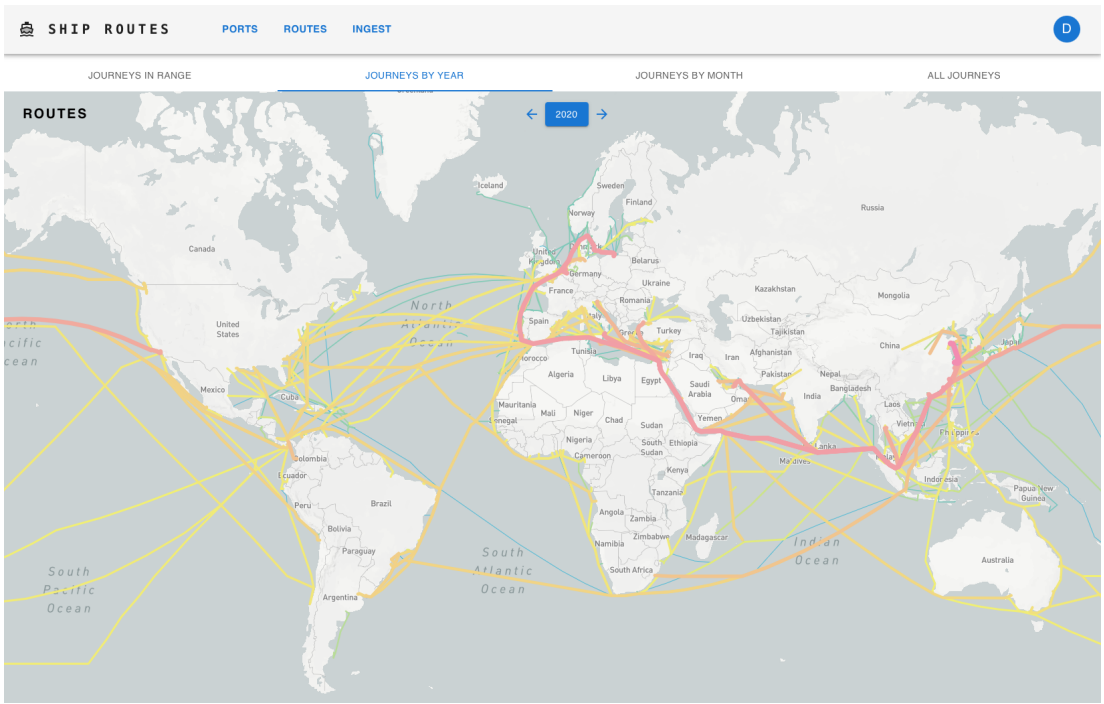


Figura 7.2: Ejemplo de las rutas marítimas en 2020

Bibliografía

[1] UNCTAD, "Review of maritime transport 2022," 2022.

[2] D. Guerrero, L. Letrouit, and C. P. Montes, "The container transport system during covid-19: An analysis through the prism of complex networks," *Transport Policy*, 2022.

[3] K. Schwaber and J. Sutherland. (2020) The scrum guide. [En línea]. Disponible en: https://scrumguides.org/scrum-guide.html

[4] R. C. Martin. (2020) Solid relevance. [En línea]. Disponible en: https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html

[5] ——, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson, 2017.

[6] ——. (2013) The clean architecture. [En línea]. Disponible en: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[7] M. Fowler. (2011) Cqrs. [En línea]. Disponible en: https://martinfowler.com/bliki/CQRS.html

[8] Baeldung. (2022) Messaging with spring amqp. [En línea]. Disponible en: https://www.baeldung.com/spring-amqp

[9] ——. (2023) A guide to the reflections library. [En línea]. Disponible en: https://www.baeldung.com/reflections-library

[10] Eurostat. Searoute. [En línea]. Disponible en: https://github.com/eurostat/searoute

[11] Mapbox guides. [En línea]. Disponible en: https://docs.mapbox.com/mapbox-gl-js/guides/

[12] Material ui. [En línea]. Disponible en: https://mui.com/material-ui/

[13] Digital ocean tutorials. [En línea]. Disponible en: https://www.digitalocean.com/community/tutorials

[14] M. Fowler. (2016) Value objet. [En línea]. Disponible en: https://martinfowler.com/bliki/ValueObject.html

[15] E. Paraschiv. (2023) Introduction to spring batch. [En línea]. Disponible en: https://www.baeldung.com/introduction-to-spring-batch

[16] Create and style clusters. [En línea]. Disponible en: https://docs.mapbox.com/mapbox-gl-js/example/cluster/